EPTCS 393

Proceedings of the

18th International Workshop on the ACL2 Theorem Prover and Its Applications

Austin, TX, USA and online, November 13-14, 2023

Edited by: Alessandro Coglio and Sol Swords

Published: 14th November 2023
DOI: 10.4204/EPTCS.393
ISSN: 2075-2180
Open Publishing Association

Table of Contents

Preface

Alessandro Coglio
Kestrel Institute & Aleo Systems, Inc.
coglio@kestrel.edu

Sol Swords
Intel Corp.
sol.swords@intel.com

This volume contains the proceedings of the Eighteenth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2023), a two-day workshop held at the University of Texas at Austin and online, on November 13-14. ACL2 workshops occur at approximately 18-month intervals, and they provide a major technical forum for users of the ACL2 theorem proving system to present research related to the ACL2 theorem prover and its applications.

ACL2 is an industrial-strength automated reasoning system, the latest in the Boyer-Moore family of theorem provers. The 2005 ACM Software System Award was awarded to Boyer, Kaufmann, and Moore for their work in ACL2 and the other theorem provers in the Boyer-Moore family.

The proceedings of ACL2-2023 include ten long papers and three extended abstracts. Each submission received three reviews. The workshop also included several "rump session" talks — short unpublished presentations that discussed ongoing research — as well as two invited talks from Jim Grundy of Amazon Web Services and Eric Smith of Kestrel Institute.

As program co-chairs we are grateful for the other leaders of this workshop: the organizing chairs, Matt Kaufmann and Mayank Manjrekar; the arrangements chair, Rob Sumners; and the registration chair, David Rager. We appreciate the great community of researchers who continue to develop exciting projects and tools using ACL2, as well as contributing papers to the workshop. We also wish to thank the program committee for helping the editorial process proceed very smoothly.

This workshop would not have been possible without the support of a large number of people. We thank those who authored, submitted, and presented papers. We also wish to thank the Program Committee for their diligence in reviewing the papers in a timely manner and for further discussions after the reviews. We are very grateful to the invited speakers for agreeing to provide their unique and extensive perspectives to the workshop participants.

Alessandro Coglio and Sol Swords, program chairs
November 2023

ACL2 2023 Program Committee:

- Matt Kaufmann, The University of Texas at Austin (retired)
- David Greve, Collins Aerospace
- Anna Slobodova, Intel Corp.
- Panagiotis Manolios, Northeastern University
- David Russinoff, Arm Inc.
- Mayank Manjrekar, Arm Inc.
- Cuong Chau, Intel Corp.
- Freek Verbeek, Open University of The Netherlands

A. Coglio, S. Swords (Eds.): 18th ACL2 Workshop (ACL2-2023)
EPTCS 393, 2023, pp. iii–iv, doi:10.4204/EPTCS.393.0

- Ruben Gamboa, University of Wyoming & Kestrel Institute
- Mitesh Jain, Rivos Inc.
- Shilpi Goel, Amazon Web Services
- Stephen Westfold, Kestrel Institute
- Mertcan Temel, Intel Corp.
- Eric McCarthy, Kestrel Institute
- David Hardin, Collins Aerospace
- Mark Greenstreet, The University of British Columbia
- Warren A. Hunt, Jr., The University of Texas at Austin
- Jared Davis, Amazon.com Services LLC

Classical LU Decomposition in ACL2

Carl Kwan
Department of Computer Science
The University of Texas at Austin
Austin, TX, USA
carlkwan@cs.utexas.edu

We present an ACL2 formalization and verification of a classical LU decomposition algorithm. LU decomposition plays a crucial role in numerical computations, linear algebra algorithms, and applications that involve matrix operations or solving systems of linear equations. However, theorem proving with matrices can be challenging because typical implementations of matrix algorithms often involve the heavy use of indexing, among other issues. Our approach to formalizing LU decomposition in ACL2 adopts a methodical and constructive derivation process that is more amenable to formal verification. We also provide an ACL2 implementation for Gaussian elimination as one of several interesting consequences of our formalization.

The broad application of LU decomposition to linear algebra, and thus science as a whole, motivates our formalization in ACL2. We are interested in building reliable numerical linear algebra software for use in the design of critical systems. To our knowledge, this is the first formalization of an LU decomposition algorithm.

Linear algebra algorithms are used widely in scientific computing, data analysis, policy and decision making, the design of reliable infrastructure, etc. Their broad applications to critical systems make linear algebra methods a prime target for formal verification. However, computational linear algebra algorithms have seen few theorem prover formalizations. On one hand, these algorithms are often expressed in terms of operations on the indexed elements of a matrix. The use of indexing can obscure the design goals of the matrix algorithm, reduce human readability, and make reasoning for theorem provers difficult, among other drawbacks. On the other hand, many theorem provers tend to have limited support for the efficient computation of operations involving numeric values. This limits the utility of formalizing particularly *computational* linear algebra methods in such theorem provers.

Targeting the need for reliable but efficient linear algebra computation, we present a formalization of an LU decomposition algorithm and verify it in ACL2. LU decomposition, also known as LU factorization, is a fundamental linear algebra algorithm that decomposes a given matrix into upper and lower triangular factors. It is a crucial step in methods that aim to efficiently solve systems of linear equations, invert matrices, and compute matrix determinants.

As part of our formalization, we also verify the derivation of the LU decomposition algorithm. The algorithm we verify is sometimes known as "classical" or "right-looking", and many mathematical texts introduce it as a sequence of Gauss transformation multiplications [1, 2, 8, 9]. Since we are using ACL2, we need to translate into recursion and induction. One typical derivation partitions a given matrix into a form that suggests recursing along the principal submatrices. However, algorithms following this derivation tend to remain heavily reliant on loops or indexing to express matrix operations. We take the more ACL2 natural approach.

An LU decomposition of a matrix A are lower and upper triangular matrices L and U, respectively, such that $A = LU$. We also make the requirement that L is unit lower triangular, i.e. L has 1s on the diagonal. In the interest of memory optimization, this extra requirement makes it possible to overwrite

A. Coglio, S. Swords (Eds.): 18th ACL2 Workshop (ACL2-2023)
EPTCS 393, 2023, pp. 1–5, doi:10.4204/EPTCS.393.1

the upper part of A with the upper part of U and the strictly lower part of A with the strictly lower part of L during the algorithm. Partition $A = LU$ as follows:

$$\left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right) = A = LU = \left(\begin{array}{c|c} 1 & \\ \hline \ell_{21} & L_{22} \end{array}\right) \left(\begin{array}{c|c} \upsilon_{11} & u_{12}^T \\ \hline & U_{22} \end{array}\right) . \tag{1}$$

Regarding notation: lower-case Greek letters are field scalars; lower-case Latin letters are vectors; upper-case Latin letters are matrices; and assume that any posed variables are "conformal", e.g. if A is $m \times n$, then a_{21} is $(m-1) \times 1$ and a_{12}^T is $1 \times (n-1)$. Moving forward, we will drop the "bars" in partitions for simplicity. Instead of looping through the rows or columns of A, the partition suggests we recurse along the principal submatrices. We want Equation (1) to hold after performing the algorithm, i.e.

$$\alpha_{11} = \upsilon_{11} , \qquad a_{21} = \upsilon_{11}\ell_{21} , \qquad a_{12}^T = u_{12}^T , \qquad A_{22} = \ell_{21}u_{21}^T + L_{22}U_{22} .$$

Since A is given, u_{12}^T and υ_{11} are obvious. Solving for the remaining components of L and U forces

$$\ell_{21} = a_{21}\alpha_{11}^{-1} , \tag{2}$$

$$L_{22}U_{22} = A_{22} - a_{21}\alpha_{11}^{-1}a_{12}^T . \tag{3}$$

This suggests an algorithm which requires merely updating a_{21} and A_{22}. Indeed, consider Algorithm 1. Since the algorithm is recursive, we need to handle a few base cases. If the matrix is empty, then we just return an empty matrix. In the case that the matrix is a column vector, then Equation (2) indicates we should return $a_{21}\alpha_{11}^{-1}$. Similarly, if we are given a row vector, then no updates are necessary. Otherwise, we can update the components of A suggested by Equations (2) and (3) and proceed with the recursion.

The ACL2 implementation of Algorithm 1 is Program 1. Descriptions of the ACL2 linear algebra functions used in this paper are in Table 1. We use existing ACL2 primitive matrix operations and functions in our formalization [3]. However, we also define both new and alternative operations that are necessary in order to implement and verify the LU decomposition algorithm. In cases where we define alternative operations, we prove they are equivalent to those already in ACL2. We defer the discussion of most of these new functions to the future, but those that appear in this paper are `out-*`, `get-U`, and `get-L`. It is also important to know that `get-L` fills the diagonal with 1s.

Algorithm 1 Classical LU decomposition

procedure LU($A \in \mathbb{R}^{m \times n}$)

Partition $A = \begin{pmatrix} \alpha_{11} & a_{12}^T \\ a_{21} & A_{22} \end{pmatrix}$

if $m = 0$ or $n = 0$ **then**

return ( )

else if $n = 1$ **then**

return $\begin{pmatrix} \alpha_{11} \\ a_{21}\alpha_{11}^{-1} \end{pmatrix}$

else if $m = 1$ **then**

return A

else

$a_{21} \coloneqq a_{21}\alpha_{11}^{-1}$

$A_{22} \coloneqq A_{22} - a_{21}a_{12}^T$

return $\begin{pmatrix} \alpha_{11} & a_{12}^T \\ a_{21} & \text{LU}(A_{22}) \end{pmatrix}$

Program 1 ACL2 implementation of Algorithm 1

```
(define lu ((A matrixp)) ...
 (b* (((unless (matrixp A)) (m-empty))
      ((if (m-emptyp A)) A)
      (alph (car (col-car A)))
      ((if (zerop alph))
       (mzero (row-count A)
              (col-count A)))
      ((if (m-emptyp (col-cdr A)))
       (row-cons (list alph)
                 (sm* (/ alph)
                      (row-cdr A))))
      ((if (m-emptyp (row-cdr A))) A)
      (a21 (col-car (row-cdr A)))
      (a12 (row-car (col-cdr A)))
      (A22 (col-cdr (row-cdr A)))
      (a21 (sv* (/ alph) a21))
      (A22 (m+ A22 (sm* -1 (out-* a21 a12)))))
     (row-cons (row-car A)
               (col-cons a21 (lu A22))))
```

With the exception of some extra edge cases, Program 1 directly implements Algorithm 1. In order for the decomposition to succeed, we require only that $\alpha_{11} \neq 0$ and $L_{22}U_{22} = A_{22} - a_{21}\alpha_{11}^{-1}a_{12}^{T}$. The ACL2 theorem for this requirement is Program 2. This requirement is interesting because, given that α_{11} is nonsingular, it reduces the condition for a matrix to be LU decomposable into a condition about a smaller matrix, which is reminiscent of some "induction step". Indeed, the RHS of Equation (3) is the *Schur complement of* α_{11} *in* A and is a key tool in statistics, probability, numerical analysis, and matrix analysis. While mathematical references commonly describe the conditions for the algorithm to succeed in terms of the leading principal submatrices, the proof that these conditions are sufficient reduces to an induction step that depends on Equation (3) [8].

In order to avoid writing an ACL2 statement about *all* the leading principal submatrices of a matrix, we recurse along the Schur complements, i.e. ensure α_{11} is nonzero and recurse on $S \coloneqq A_{22} - a_{21}\alpha_{11}^{-1}a_{12}^{T}$. This recognizes matrices with nonsingular leading principal submatrices. Indeed, consider Program 3. The function `nonsingular-leading-principal-submatrices-p` only returns `nil` on a matrix if it encounters an α_{11} that is zero. One way to think about this condition is that if no zeros appear after k recursive steps, then the k-th leading principal submatrix is nonsingular because its determinant is nonzero. Instead of reasoning with determinants, which are rarely useful in numerical algorithms [9], Schur complements provide a more concise ACL2 condition for success and presents opportunites for further rich ACL2 explorations in numerical linear algebra.

To prove the correctness of the classical LU decomposition for a square matrix, we want to assume that the matrix has nonsingular leading principal submatrices and induct along their Shur complements. In ACL2, these are both satisfied by assuming `nonsingular-leading-principal-submatrices-p` from Program 3 in the hypothesis. Indeed, as seen in Program 4, including such a hypothesis enables ACL2 to prove the desired theorem without any user-provided hints. ACL2 automatically inducts according to a scheme suggested by `nonsingular-leading-principal-submatrices-p`, the induction step for which enables the use of `lu-correctness-induction-step` from Program 2.

Table 1 ACL2 linear algebra functions

Function	*Description*
`matrixp`	Recognizer for matrices
`m-emptyp`	Recognizer for empty matrices
`m-empty`	Returns an empty matrix
`mzero`	Returns a zero matrix
`row-car`	Returns first row of a matrix
`col-car`	Returns first column of a matrix
`row-cdr`	Remove a matrix's first row
`col-cdr`	Remove a matrix's first column
`row-cons`	Append a row to a matrix
`col-cons`	Append a column to a matrix
`m+`	Matrix addition
`m*`	Matrix multiplication
`sm*`	Scalar-matrix multiplication
`sv*`	Scalar-vector multiplication
`out-*`	Outer product of vectors
`get-L`	Get a matrix's lower triangular part
`get-U`	Get a matrix's upper triangular part

Program 2 ACL2 LU "induction step" correctness

```
(defthm lu-correctness-induction-step
 (b* ((alph (car (col-car A)))
      (LU   (lu A))
      (L    (get-L LU))
      (U    (get-U LU))
      (a21  (col-car (row-cdr A)))
      (a12  (row-car (col-cdr A)))
      (A22  (col-cdr (row-cdr A)))
      (a21  (sv* (/ alph) a21))
      (A22  (m+ A22 (sm* -1 (out-* a21 a12)))))
     (implies (and (matrixp A)
                   (not (m-emptyp (row-cdr A)))
                   (equal (col-count A)
                          (row-count A))
                   (not (zerop alph))
                   (equal (m* (get-L (lu A22))
                              (get-U (lu A22)))
                          A22))
              (equal (m* L U) A))) ...)
```

Program 3 ACL2 recognizer for matrices with nonsingular leading principal submatrices

```
(define nonsingular-leading-principal-submatrices-p ((A matrixp))
 :measure (and (row-count A) (col-count A))
 (b* (((unless (matrixp A)) nil)
      ((if (m-emptyp A)) t)
      (alph (car (col-car A)))
      ((if (zerop alph)) nil)
      ((if (or (m-emptyp (row-cdr A))
               (m-emptyp (col-cdr A))))
       t)
      ;; Compute S = A22 - out-*(a21/alph,a12)
      (a21   (col-car (row-cdr A)))
      (a12   (row-car (col-cdr A)))
      (A22   (col-cdr (row-cdr A)))
      (a21/a (sv* (/ alph) a21))
      (S     (m+ A22 (sm* -1 (out-* a21/a a12)))))
     (nonsingular-leading-principal-submatrices-p S)) ...)
```

Program 4 ACL2 LU correctness

```
(defthm lu-correctness
 (b* ((LU (lu A))
      (L  (get-L LU))
      (U  (get-U LU)))
     (implies (and (equal (col-count A) (row-count A))
                   (nonsingular-leading-principal-submatrices-p A))
              (equal (m* L U) A))))
```

There are many applications to our formalization. One such application is that we can now calculate the determinant of an LU decomposable matrix A since $\det(A) = \det(L)\det(U) = \det(U)$ is simply the product of the diagonal of U. Another consequence of Algorithm 1 is that the computed U is the row echelon form of A. In particular, this means that `(defun ge (A) (get-U (lu A)))` is the ACL2 formalization of Gaussian elimination, which is one step away from an ACL2 formalization of Gauss-Jordan elimination. Gauss-Jordan can be used to find matrix inverses or even solve systems of linear equations such as $Ax = b$.

In practice, it tends to be better to use LU decomposition over forming an inverse. A linear system $Ax = b$ can be solved by forming L and U, and then solving $Ly = b$ and $Ux = y$ via forwards and backwards substitution, respectively. This is faster than using Gauss-Jordan to form an inverse [8]. We have formalized backwards and forwards substitution in ACL2, the discussion for which we defer to the future; with LU decomposition, they form an ACL2 method for solving systems of linear equations.

Previously, we formalized the Cauchy-Schwarz inequality and other vector-related ideas from linear algebra and convex optimization [6, 5, 4, 7]. Here we show that our formalization of LU decomposition in ACL2 leads to a rich selection of research directions. We look forward to formalizing more methods for solving systems of linear equations. In addition, there are other decompositions (QR, rank-reducing, spectral, etc.), optimization methods (least-squares, gradient descent, etc.), and matrix analysis ideas (norms, tensors, etc.) to explore. These directions all serve as important methods in pure and applied mathematics, science, and engineering, and we expect to formalize verified and efficient computational linear algebra methods in the future.

References

[1] Robert van de Geijn & Margaret Myers (2023): *Advanced Linear Algebra: Foundations to Frontiers*. Available at `https://www.cs.utexas.edu/users/flame/laff/alaff/frontmatter.html`.

[2] Gene H. Golub & Charles F. Van Loan (2013): *Matrix Computations - 4th Edition*. Johns Hopkins University Press, Philadelphia, PA, doi:10.56021/9781421407944.

[3] Joe Hendrix (2003): *Matrices in ACL2*. Available at `https://www.cs.utexas.edu/users/moore/acl2/workshop-2003/contrib/hendrix/hendrix.pdf`.

[4] Carl Kwan (2019): *Automated reasoning in first-order real vector spaces*. Master's thesis, University of British Columbia, doi:10.14288/1.0380600. Available at `https://open.library.ubc.ca/collections/ubctheses/24/items/1.0380600`.

[5] Carl Kwan & Mark R. Greenstreet (2018): *Convex Functions in ACL2(r)*. In Shilpi Goel & Matt Kaufmann, editors: Proceedings of the 15th International Workshop on the *ACL2 Theorem Prover and Its Applications*, Austin, Texas, USA, November 5-6, 2018, *Electronic Proceedings in Theoretical Computer Science* 280, Open Publishing Association, pp. 128–142, doi:10.4204/EPTCS.280.10.

[6] Carl Kwan & Mark R. Greenstreet (2018): *Real Vector Spaces and the Cauchy-Schwarz Inequality in ACL2(r)*. In Shilpi Goel & Matt Kaufmann, editors: Proceedings of the 15th International Workshop on the *ACL2 Theorem Prover and Its Applications*, Austin, Texas, USA, November 5-6, 2018, *Electronic Proceedings in Theoretical Computer Science* 280, Open Publishing Association, pp. 111–127, doi:10.4204/EPTCS.280.9.

[7] Carl Kwan, Yan Peng & Mark R. Greenstreet (2020): *Cauchy-Schwarz in ACL2(r) Abstract Vector Spaces*. In Grant Passmore & Ruben Gamboa, editors: Proceedings of the Sixteenth International Workshop on the *ACL2 Theorem Prover and its Applications*, Worldwide, Planet Earth, May 28-29, 2020, *Electronic Proceedings in Theoretical Computer Science* 327, Open Publishing Association, pp. 90–92, doi:10.4204/EPTCS.327.8.

[8] G. W. Stewart (1998): *Matrix Algorithms*. Society for Industrial and Applied Mathematics, doi:10.1137/1.9781611971408.

[9] Lloyd N. Trefethen & David Bau, III (1997): *Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, Philadelphia, PA, doi:10.1137/1.9780898719574.

CHERI Concentrate in ACL2*

Carl Kwan Yutong Xin William D. Young
Department of Computer Science
The University of Texas at Austin
Austin, TX, USA
{carlkwan, maxxin, byoung}@cs.utexas.edu

We present an ACL2 formalization of CHERI Concentrate, a practical compression format for capabilities. Capabilities are unforgeable tokens of authority that grant and describe access to a region of memory; CHERI Concentrate enables the efficient storage and encoding / decoding of capabilities in real-life reliable hardware. As part of our formalization, we verify the correctness of the encoding and decoding functions, which specify the bounds in memory a program is permitted to access. These encode / decode functions are complicated and their correctness is not obvious to a human reader. This along with the use of capabilities in purportedly secure hardware is why formal verification is important. To our knowledge, this is the first executable formalization of CHERI Concentrate in ACL2.

We present a formalization of CHERI-style capabilities in ACL2. CHERI (Capability Hardware Enhanced RISC Instructions) is a set of architectural features that extends conventional hardware Instruction-Set Architectures (ISAs) with capabilities that enable fine-grained memory protection and highly scalable software compartmentalization [7]. Capabilities are hardware descriptions for permissions that can be used to access data, code, and objects in a controlled and protected manner. Existing ISAs extended with CHERI-style capabilities include 64-bit MIPS, 32-bit RISC-V, 64-bit RISC-V, and 64-bit Armv8-A [6]. More recently, Arm has shipped a CHERI-enabled Morello prototype processor, SoC, and board [1, 2]. The work we present here is part of an ongoing effort to model CHERI capabilities in ACL2 for y86 and, eventually, x86 ISAs.

In CHERI systems, any accesses to memory are managed by capabilities. In particular, a capability will store in memory the upper and lower bounds of a region that a program is permitted to access. The capabilities that interest us are 128-bit data structures that can be stored in memory or registers. However, in a 64-bit system, this means that the bounds, which are two 64-bit addresses, are encoded in a 128-bit capability; this is in addition to other information that a capability must describe (e.g. permissions). Storing all pertinent information uncompressed can require 256 bits, which enacts a heavy toll on memory and quickly saturates a system's cache and data-paths [6].

CHERI Concentrate is a format that compresses capabilities to 128 bits on 64-bit architectures and reduces L2 cache misses by 50% to 70% compared to 256-bit capabilities [8]. This format compresses the bounds to 27 bits compared to 128 bits uncompressed. However, such a reduction in space requires a complicated encoding algorithm – sufficiently complicated that it is not clear from their definitions whether the encode / decode functions perform the desired tasks. In fact, initial iterations of the encoding algorithm were erroneous and required the use of HOL4, a higher-order logic proof assistant, to guide the development of the compression algorithm. The complexity of CHERI Concentrate warrants the use of formal methods, but HOL4 has limited support for execution.

In addition to formalizing CHERI capabilities, we verify the correctness of the CHERI Concentrate encoding and decoding algorithms in ACL2. By performing our formalization in ACL2, we can take

*This work was supported in part by SRI International and the Defense Advanced Research Projects Agency of the United States Department of Defense.

A. Coglio, S. Swords (Eds.): 18th ACL2 Workshop (ACL2-2023)
EPTCS 393, 2023, pp. 6–10, doi:10.4204/EPTCS.393.2

advantage of ACL2's capacity for reasoning and execution efficiency. Moreover, given the size and complexity of the CHERI ISA in general, future CHERI models in ACL2 would rely on our formalization of CHERI Concentrate every time a capability is called. Indeed, we are currently building an ACL2 model of the y86-64 ISA with CHERI-style capabilities with the intent of verifying machine code programs involving capabilities. Since our y86-64 model symbolically simulates the execution of machine code, our formalization of CHERI Concentrate needs to be executable in order to be useful. Using the ACL2 y86-64 model extended with capabilities in CHERI Concentrate format, we can verify machine code programs involving capabilities, though this is beyond the scope of this paper and we intend on sharing these results at a later date. To our knowledge, this is the first executable formalization of CHERI Concentrate in ACL2.

Space limitations allow only a brief description here of 128-bit CHERI capabilities and the CHERI Concentrate format. For details, an interested reader may refer to the CHERI ISA [6]. One method of motivating the introduction of CHERI capabilities is to consider them as *fat pointers*, i.e. a pointer extended with additional metadata such as bounds and permissions that may restrict the pointer's behaviour. In principle, a pointer contains an address that can be dereferenced in order to access a region of memory indicated by the address. However, in the interest of memory protection, the metadata associated with a fat pointer may store permissions to restrict how memory may be accessed and bounds to restrict the region of memory that may be accessed. CHERI design principles further restrict how capabilities may be constructed and updated, requiring additional metadata not typically found in traditional fat pointers.

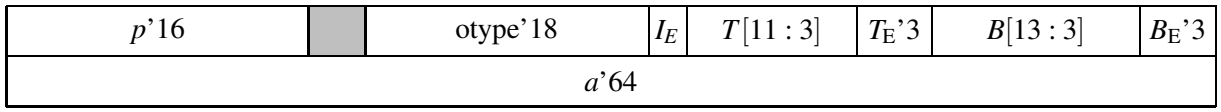
We make a distinction between *architectural capabilities* and capabilities in memory. Architectural capabilities contain software-accessible fields that are not reflected explicitly by, but can still be inferred from, a given capability's in-memory representation. In our model, architectural capabilities contain the following fields: 1-bit validity tag, 16-bit permissions (perms), 18-bit object type (otype), 64-bit offset, 64-bit base, and 64-bit length. The base is the lower bound of the memory region dereferenceable by a capability. Adding length to base gives the upper bound (a.k.a. top) of the memory region dereferenceable by a capability. Adding offset to base gives the address when the capability is used as a pointer.

System constraints necessitate compressing the information provided by an architectural capability into the format described by CHERI Concentrate, which is then stored in memory. As part of our formalization, we develop and verify ACL2 functions that allow a user to easily convert between architectural and memory-resident capabilities. The proofs of these conversion functions amount to verifying that converting from architectural capabilities to memory-resident capabilities and then back to architectural capabilities recovers the information that was originally in the architectural capabilities, and vice-versa.

A visual representation of memory-resident capabilities in CHERI Concentrate format can be seen in Figure 1. In the diagram, a is a 64-bit address, p represents 16 bits for the permissions, $otype$ represents 18 bits for the object type, and 27 bits are used to encode bounds relative to the address. The compression uses a floating-point representation to encode the bounds. Regarding notation for the bounds: B and T are 14-bit values that form the base and top of a capability's bounds relative to its address (the top two bits of T are omited during encoding but deduced from B during decoding); I_E is an internal exponent bit that, when set, indicates the lower three bits of B and T (B_E and T_E, respectively) represents an exponent at the expense of three bits of precision; E is a 6-bit exponent that determines the position at which B and T are inserted into a.

The bulk of our verification efforts is devoted to proving the encoding and decoding functions correct. As part of the compression process, CHERI Concentrate specifies the algorithms by which the bounds are encoded into B and T, and how B and T can be decoded. To illustrate the complexity of these algorithms, consider Listing 1, which is the ACL2 implementation of the decoding algorithm `decode-compression`. We omit the similarly complicated function `encode-compression` for brevity.

Figure 1: CHERI Concentrate format



Listing 1: Decoding function in ACL2

```
(define decode-compression ((c compressionp) (addr (nbp addr *xlen*))) ...
  (b* ((IE (nb-ash (+ *mw* *tw*) 1 c))
       ((mv E TS BL Lcarry Lmsb)
        (if (= IE 0)
            (vars-small-seg c)
            (vars-large-seg c)))
       (BLH (+ (nb-ash *tw* (- *mw* *tw*) BL) Lcarry Lmsb))
       (TLH (nb-ash 0 (- *mw* *tw*) BLH))
       (TL (logior (ash TLH *tw*) TS))
       ((mv CT CB) (corrections addr TL BL E))
       (AT (ash addr (- (+ *mw* E))))
       (top (nb-ash 0 (1+ *xlen*) (ash (logior (ash (+ AT CT) *mw*) TL) E)))
       (base (nb-ash 0 *xlen* (ash (logior (ash (+ AT CB) *mw*) BL) E)))
       (top (if (and (< E 51)
                     (< 1 (- (nb-ash 63 2 top) (nb-ash 63 1 base))))
                (nb-ash 0 *xlen* top)
                top)))
    (bounds top base)))
```

The function `encode-compression` compresses a base b_0 and length ℓ_0, that together form a top $t_0 = b_0 + \ell_0$, into a format reflected by the portion of Figure 1 that pertains to I_E, T, and B. The function `decode-compression` reconstructs bounds relative to an address from the compression to obtain a base b_1 and top t_1 that may have been subjected to some rounding. Again, the interested reader can find motivating details in the CHERI ISA and the original CHERI Concentrate paper [6, 8]. The properties of these functions that we have verified are:

1. $b_0 \geq b_1$ for any b_0, t_0, and address;
2. $b_0 - b_1 \leq 2^{E+3}$ for any b_0, t_0, and address;
3. $t_0 \leq t_1$ for any b_0, t_0, and address;
4. $t_1 - t_0 \leq 2^{E+3}$ for any b_0, t_0, and address;
5. $b_0 = b_1$ when $\ell_0 < 2^{12}$ or when the lower $E+3$ bits of b_0 and t_0 are zero;
6. $t_0 = t_1$ when $\ell_0 < 2^{12}$ or when the lower $E+3$ bits of b_0 and t_0 are zero.

The last two properties state the conditions under which decoding a compression will recover the exact bounds that were originally encoded, which is important because compression may result in loss of precision. These last two properties contain stronger conclusions than the other properties and were not proven in the original CHERI Concentrate paper.

Our approach to proving these properties makes heavy use of the symbolic simulation framework GL with case-splitting [5, 3]. While discharging these proofs by introducing traditional ACL2 rewrite rules would of course be feasible, we find that it takes ACL2 over 90 minutes to verify the returns specifier of a capability constructor without user-provided hints. This suggests that verification of the encode / decode

Listing 2: Theorem of encode / decode property 1 in ACL2

```
(def-gl-param-thm decode-encode-b-bound-len>2^12
    :hyp   (and (valid-addr-p addr base len)
                (valid-b-l-p base len)
                (<= (expt 2 *tw*) len))
    :concl (<= (bounds->base (decode-compression (encode-compression len base) addr))
               base)
    :param-bindings
    `((((low 12) (high 16)) ,(gl::auto-bindings (:mix (:nat base 65) (:nat len 65) (:nat addr 65))))
      (((low 16) (high 20)) ,(gl::auto-bindings (:mix (:nat base 65) (:nat len 65) (:nat addr 65))))
      (((low 20) (high 24)) ,(gl::auto-bindings (:mix (:nat base 65) (:nat len 65) (:nat addr 65))))
      (((low 24) (high 28)) ,(gl::auto-bindings (:mix (:nat base 65) (:nat len 65) (:nat addr 65))))
      (((low 28) (high 32)) ,(gl::auto-bindings (:mix (:nat base 65) (:nat len 65) (:nat addr 65))))
      (((low 32) (high 36)) ,(gl::auto-bindings (:mix (:nat base 65) (:nat len 65) (:nat addr 65))))
      (((low 36) (high 40)) ,(gl::auto-bindings (:mix (:nat base 65) (:nat len 65) (:nat addr 65))))
      (((low 40) (high 44)) ,(gl::auto-bindings (:mix (:nat base 65) (:nat len 65) (:nat addr 65))))
      (((low 44) (high 48)) ,(gl::auto-bindings (:mix (:nat base 65) (:nat len 65) (:nat addr 65))))
      (((low 48) (high 52)) ,(gl::auto-bindings (:mix (:nat base 65) (:nat len 65) (:nat addr 65))))
      (((low 52) (high 56)) ,(gl::auto-bindings (:mix (:nat base 65) (:nat len 65) (:nat addr 65))))
      (((low 56) (high 60)) ,(gl::auto-bindings (:mix (:nat base 65) (:nat len 65) (:nat addr 65))))
      (((low 60) (high 64)) ,(gl::auto-bindings (:mix (:nat base 65) (:nat len 65) (:nat addr 65))))
      (((low 64) (high 65)) ,(gl::auto-bindings (:mix (:nat base 65) (:nat len 65) (:nat addr 65)))))
    :param-hyp (and (<= (expt 2 low) len) (< len (expt 2 high)))
    :cov-bindings (gl::auto-bindings (:mix (:nat base 65) (:nat len 65) (:nat addr 65))))
```

properties would be better suited to more automatic methods. Moreover, since these are essentially bittwiddling proofs, bitblasting would be an amenable approach.

GL supports model checking with both BDDs as ACL2 symbolic objects and external SAT solvers. We use BDDs as the back-end engine for GL. While case splits were necessary for some of the larger proofs, we found that GL BDDs were sufficient for our verification needs. As such, we did not utilize any external reasoning engine nor FGL, the successor to GL [4]. Moreover, GL's BDD proof procedure is verified in ACL2, alleviating any extra soundness concerns that may arise from calling external tools.

The ACL2 proofs of the encode / decode properties 5-6 are straightforward applications of GL's default symbolic simulation event `def-gl-thm`. In contrast, properties 1-4 all required parameterized case splitting. Listing 2 is the ACL2 theorem of the encode / decode property 1. As indicated by the `:param-bindings` and `:param-hyp`, we split the theorem into 4-bit "intervals", effectively performing individual symbolic simulations for when the length ℓ_0 satisfies $2^{12} \leq \ell_0 < 2^{16}$ and $2^{16} \leq \ell_0 < 2^{20}$ and so on. Case splits for properties 2-4 are the same. We found that case splitting into smaller bit "intervals" would discharge the proofs faster at the expense of verbose user-provided hints. Case splitting being an effective approach suggests that the complexity of encoding and decoding does not scale with the length of the memory regions. Indeed, this corroborates with the design principles of CHERI Concentrate: "encoding efficiency, minimize delay of pointer arithmetic, and eliminate additional load-to-use delay" [8]. On the other hand, properties 5-6 required no case splitting largely because the encode / decode logic is greatly simplified when $\ell_0 < 2^{12}$, and alignment requirements reduce the number of variable bits.

Arm is deploying CHERI-enabled devices to industry security specialists at companies such as Google and Microsoft; they describe CHERI hardware capabilities as "a fundamentally more secure building block for software" [2]. It is critical that functions controlling memory accesses in these secure systems are correct; we have demonstrated that ACL2 is a excellent tool for modelling CHERI capabilities and verifying CHERI functions. While other formal methods have been part of the toolbox in developing CHERI-extended ARM, RISC, and MIPS ISAs, the application of CHERI to the Intel x86-64 ISA remains a sketch and has not yet been implemented. This presents an opportunity, which we intend to pursue, to perform ACL2 formal modelling and verification in the CHERI-x86-64 space.

References

[1] Saar Amar (2022): *An Armful of CHERIs*. Available at https://msrc.microsoft.com/blog/2022/01/an_armful_of_cheris/.

[2] Richard Grisenthwaite (2022): *Morello research program hits major milestone with hardware now available for testing*. Available at https://www.arm.com/company/news/2022/01/morello-research-program-hits-major-milestone-with-hardware-now-available-for-testing.

[3] Sol Swords (2017): *Term-Level Reasoning in Support of Bit-blasting*. In Anna Slobodova & Warren Hunt, Jr., editors: Proceedings 14th International Workshop on the *ACL2 Theorem Prover and its Applications*, Austin, Texas, USA, May 22-23, 2017, *Electronic Proceedings in Theoretical Computer Science* 249, Open Publishing Association, pp. 95–111, doi:10.4204/EPTCS.249.7.

[4] Sol Swords (2020): *New Rewriter Features in FGL*. In Grant Passmore & Ruben Gamboa, editors: Proceedings of the Sixteenth International Workshop on the *ACL2 Theorem Prover and its Applications*, Worldwide, Planet Earth, May 28-29, 2020, *Electronic Proceedings in Theoretical Computer Science* 327, Open Publishing Association, pp. 32–46, doi:10.4204/EPTCS.327.3.

[5] Sol Swords & Jared Davis (2011): *Bit-Blasting ACL2 Theorems*. In David Hardin & Julien Schmaltz, editors: Proceedings 10th International Workshop *on the ACL2 Theorem Prover and its Applications*, Austin, Texas, USA, November 3-4, 2011, *Electronic Proceedings in Theoretical Computer Science* 70, Open Publishing Association, pp. 84–102, doi:10.4204/EPTCS.70.7.

[6] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son & Hongyan Xia (2020): *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8)*. Technical Report 951, University of Cambridge Computer Laboratory. Available at https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-951.pdf.

[7] Robert N.M. Watson, Simon W. Moore, Peter Sewell & Peter G. Neumann (2019): *An Introduction to CHERI*. Technical Report 941, University of Cambridge Computer Laboratory. Available at https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-941.pdf.

[8] Jonathan Woodruff, Alexandre Joannou, Hongyan Xia, Anthony Fox, Robert Norton, David Chisnall, Brooks Davis, Khilan Gudka, Nathaniel Filard, A. Theodore Makettos, Michael Roe, Peter G. Neumann, Robert N.M. Watson & Simon W. Moore (2019): *CHERI Concentrate: Practical Compressed Capabilities*. *IEEE Transactions on Computers* 68(10), pp. 1455–1469, doi:10.1109/TC.2019.2914037.

A Bound-Finding Tool for Arithmetic Terms

Sol Swords
Intel Corp.
sol.swords@intel.com

We describe a tool for establishing and proving upper and lower bounds for ACL2 arithmetic terms. The core algorithm used in this tool computes bounds of arithmetic terms using abstract interpretation. While this core algorithm is not new, we combine it with phased simplification using ACL2's rewriter, case splitting, and use of existing rules and user hints to provide bounds for subterms. Users may first expand and simplify a term as needed, split into cases if necessary, then find bounds for the simplified term using hypotheses, linear rules, type reasoning, and the core abstract interpretation alorithm. Finally, the same steps are retraced in order to prove the theorem that these bounds hold of the original term.

1 Introduction

Verification of arithmetic algorithms sometimes involves proving upper and lower bounds for intermediate values, and in many cases it isn't clear what those bounds need to be. Generally we'd prefer to prove the narrowest bounds possible in order to have the best chance of succeeding in the rest of the proof. For example, an iterative approximation algorithm for square root might converge as long as each partial remainder is smaller than some threshold, but the verifier (and, indeed, the designer) may not know exactly what that threshold is. With some study, it may be possible to determine the bound that is needed to allow the rest of the proof to go through. But it is often easier and more pragmatic to prove the best bound you can with some limited level of effort and move on to see whether this bound is sufficient for the rest of the proof.

To support this pragmatic approach, we provide a tool `def-bounds` that quickly determines and proves upper and lower bounds for an ACL2 arithmetic term. The tool allows rewriting the input term and optionally splitting into cases, then running a core algorithm that attempts to find bounds for the resulting term by a form of abstract interpretation very similar to that of Moore's Tau bounders and associated Ainni tool [4]. The core bounding algorithm is verified so that it can be used within an ACL2 metafunction. Initially, the `def-bounds` tool first emulates the given sequence of rewriting and case splitting steps that the prover will ultimately perform, then applies the bounding algorithm to find the bounds on the resulting rewritten term. It then proves these bounds by emitting a `defthm` form with a series of hints replicating the same rewriting and case splitting steps, followed by the application of this metafunction to complete the proof.

It might be logical to ask why we don't use ACL2's built-in nonlinear arithmetic procedure [1] to find bounds. This idea is appealing because this nonlinear arithmetic procedure is quite powerful, but unfortunately it has no built-in ability to find a bound that it can prove. We could search for such a bound by trial and error, but this is complicated by the fact that this nonlinear procedure can be quite slow when several multiplications are involved. We have encountered problems that nonlinear arithmetic fails to solve after 20 minutes on a modern CPU, even when our less-complete bounding algorithm can prove them. On the other hand, `def-bounds` simply produces the best bounds that it can prove with a linear

A. Coglio, S. Swords (Eds.): 18th ACL2 Workshop (ACL2-2023)
EPTCS 393, 2023, pp. 11–15, doi:10.4204/EPTCS.393.3

pass over the input term; when needed, greater accuracy can sometimes be attained by case-splitting to consider sub-ranges of values for certain subterms.

2 Example

A simple example of the operation of def-bounds follows. We define a function `foo` and try to derive bounds for it given bounds for its input `x`:

```
(include-book "centaur/misc/def-bounds" :dir :system)

(defund foo (x)
  (- (* x x) (* 3 x)))

(def-bounds foo-bounds
  (foo x)
  :hyp (and (rationalp x)
            (<= 2 x)
            (<= x 4))
  :simp-hints ((:in-theory (enable foo))))
```

The above event first runs the bound-finding routine and derives bounds $[-8,10]$ for `(foo x)`, then emits a `defthm` event to prove them, producing a theorem called `foo-bounds`. These bounds aren't very precise (the actual range is $[-2,4]$), because the two subterms `(* x x)` and `(- (* 3 x))` are considered independently and their respective bounds are $[4,16]$ and $[-12,-6]$. Somewhat better bounds can be obtained by instead bounding the factorization `(* x (- x 3))`, as follows:

```
(defthmd my-factor
  (equal (+ (- (* 3 x)) (* x x))
         (* x (- x 3))))

;; this rule would reverse the application of my-factor
(local (in-theory (disable distributivity)))

(def-bounds foo-better-bounds
  (foo x)
  :hyp (and (rationalp x)
            (<= 2 x)
            (<= x 4))
  :simp-hints ((:in-theory (enable foo))
               (:in-theory (enable my-factor))))
```

This now instead has two simplifcation phases, first allowing `foo` to open and subsequently to apply `my-factor`. Here the bounds for the two factors are $[2,4]$ and $[-1,1]$, so the computed bounds for the term are $[-4,4]$. A further option for narrowing the computed range is discussed in Section 4.

3 Core Algorithm

The core algorithm used by `def-bounds` is very similar to Ainni [4] in that it recurs through the arithmetic operators of a term, gets the bounds for the subterms, if possible, and calls a specialized routine for each arithmetic operator to determine the bounds of the result given the bounds of the operands. A

minor but useful refinement is that we add special treatment for square forms: we provide an automatic lower bound of 0 for terms matching `(* x x)`, and additionally treat terms matching `(* x x y)` as `(* (* x x) y)` so that we get this benefit for the squared portion. On the other hand, we omit, for now, Ainni's support for bitwise operators and `mod`; we also treat all bounds as weak bounds rather than distinguishing between less than and less than or equal. Another difference is in the methods that we use to determine bounds in the base case where the term is not an arithmetic operator. Ainni is aimed at the particular problem of simplifying reads of writes in a byte-addressed memory, so it only supports calls of a function R (read bytes from a memory address) as its base case, both by looking for type-alist entries (assumptions) that establish known bounds for such terms and by using basic facts about the R function (i.e., a read of n bytes is between 0 and $2^{8n} - 1$). Our core algorithm uses linear rules, typeset reasoning, and user suggestions (checked by backchaining) to find candidate bounds for all subterms. We briefly describe each of these sources of bounds.

First, we consider user-provided suggestions. The user interface for this allows suggestions of the following forms:

```
(< a 3)                        ;; upper-bound the term A by 3
(>= (foo b) 5)                 ;; lower-bound the term (FOO B) by 5
(:free (c) (<= (bar c) 10))    ;; upper-bound any call of BAR by 10
```

These are processed into a triple consisting of an LHS pattern (which may have free and/or fixed variables), an RHS term, and a direction, which together have the meaning "for subterms matching LHS, try to prove them (less/greater) than RHS." In this case "matching" means that a term unifies with LHS. The RHS may then contain some or all of the variables of LHS, which are then replaced with their matches in the term. This can be used as a bound as long as the result of this substitution is a ground term evaluating to a rational number and we can verify its correctness. We use the `mfc-relieve-hyp` utility to check the validity of the suggested bound; this tries to prove this inequality by backchaining in ACL2's rewriter, as though it appeared in the hypothesis of a rewrite rule. The result of `mfc-relieve-hyp` can be trusted in the context of the the algorithm's proof of correctness due to ACL2's meta-extract feature [3][2].

Second, we call ACL2's typeset reasoning function `mfc-ts` to obtain a typeset for the term. This typeset is an integer in which each bit position signifies a certain type of object; an unset (0) bit in a given position signifies that the term cannot be of the corresponding type. Among these basic types corresponding to bits in the typeset are negative integer, 0, 1, integer greater than 1, negative ratio, and positive ratio. If the only possible types for the term are among these, then we can sometimes determine bounds on that basis; if e.g., the only bits set in our typeset are for 0 and negative-ratio, then this gives us an upper bound of 0. Again, the meta-extract feature allows us to assume that the typeset returned by `mfc-ts` is correct so that this determination can be trusted.

Third, we consider enabled linear arithmetic theorems where either side of the inequality matches our term. For each such theorem, we check that the other side of the inequality is a ground term that reduces to a rational number after applying the unifying substitution, that this number is a better bound than the current best one found, and that the hypotheses of the theorem can be relieved by backchaining, i.e. `mfc-relieve-hyp`. Since meta-extract allows us to assume the correctness of existing linear lemmas and of `mfc-relieve-hyp`, this is sufficient to allow this bound to be trusted once found.

Fourth, if the term is an application of any of the supported arithmetic operators, then we try to get bounds in the Ainni style, by recursively bounding the arguments and applying that operator's bounding routine to those inputs. The least upper bound and greatest lower bound produced by all four methods is then returned.

4 Preprocessing

While the core bound-finding algorithm described above is powerful, by itself it has a couple of deficiencies that the `def-bounds` tool addresses with extra features.

First, the bounding algorithm isn't very convenient to use by itself. A very basic issue is that while it will prove bounds of an arithmetic term, often we want to state a theorem about a function that expands to an arithmetic term instead of the term itself. Another issue is that a given arithmetic term might not be in the best form for finding bounds. As a simple example, suppose we want bounds for $ac+bc$ and while we have bounds for a, b, and c, we have better bounds for $d = a+b$ than our bounding algorithm can determine from the bounds for a and b separately. Then it is better to rewrite this expression to dc before applying the bound-finding algorithm. The `def-bounds` tool supports this sort of need by allowing the user to provide a series of ACL2 hint objects, each giving instructions for a phased simplification step. For example, a `def-bounds` form might include the following argument, describing two simplification phases in sequence in which first `my-function` is expanded and then $ac+bc$ is rewritten to dc:

```
:simp-hints
((:expand ((my-function x y)))
 (:in-theory (enable rewrite-ac-plus-bc-to-dc)))
```

A second deficiency is due to the bounding algorithm's separate consideration of correlated subterms. The example in Section 2 shows how this may make the computed bounds imprecise; in that case the subterms `(* x x)` and `(- (* 3 x))` were considered separately with their ranges simply summed, producing bounds $[-8,10]$ where the actual range is $[-2,4]$. We saw that expressing the same function in a different form—in this case factored form rather than distributed—produced much better results, though still not perfectly precise. Another way to address this problem is to make the bounds more accurate by separately considering various ranges for the subexpressions. For example, if we split the range of x into two segments $[2,3]$ and $[3,4]$, we get much better results on the original (distributed, not factored) formulation. For $x \in [2,3]$, the algorithm concludes that the bounds for the two subexpressions x^2 and $3x$ are $[4,9]$ and $[6,9]$, giving $[-5,3]$ for the full expression; and for $x \in [3,4]$, the bounds for the two subexpressions are $[9,16]$ and $[9,12]$, giving $[-3,7]$. Taking the union of those two ranges, this gives bounds $[-5,7]$—still somewhat imprecise but greatly improved over the result without the case split. On the factored form, we get $[-3,0]$ as the range when x is in $[2,3]$ and $[0,4]$ when x is in $[3,4]$, so the bounds computed for the full domain are $[-3,4]$; again somewhat more precise than the bounds computed for the factored form without a casesplit.

Given the speed of the core algorithm, it can be practical to split into many such cases if doing so increases the accuracy of the result. For example, the above problem may be split into 128 subranges with the `def-bounds` event still completing in a few seconds. For the non-factored form this results in computed bounds $[-131/64, 259/64]$, which is only $65/64$ the size of the actual range, and for the factored form the computed bounds are $[-129/64, 4]$, even slightly better. This can be done simply by adding the following argument to the `def-bounds` form:

```
:cases ((:ranges-from-to-by x 2 4 1/64))
```

5 Conclusion

The `def-bounds` tool is in active use at Intel, especially in a successful verification of an RTL implementation of floating point square root. It is available under an MIT license from the public ACL2 books repository and its user-level documentation is available at the ACL2 documentation webpage [5].

References

[1] Warren A. Hunt, Robert Bellarmine Krug & J. Moore (2003): *Linear and Nonlinear Arithmetic in ACL2*. In Daniel Geist & Enrico Tronci, editors: *Correct Hardware Design and Verification Methods*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 319–333, doi:10.1007/978-3-540-39724-3_29.

[2] Matt Kaufmann (Accessed: 2023): *ACL2 documentation: Meta-extract*. Available at `https://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2____META-EXTRACT`.

[3] Matt Kaufmann & Sol Swords (2017): *Meta-extract: Using Existing Facts in Meta-reasoning*. *Electronic Proceedings in Theoretical Computer Science* 249, p. 47–60, doi:10.4204/eptcs.249.4.

[4] J Strother Moore (2017): *Computing Verified Machine Address Bounds During Symbolic Exploration of Code*. In Mike Hinchey, Jonathan P. Bowen & Ernst-Rüdiger Olderog, editors: *Provably Correct Systems*, Springer International Publishing, Cham, pp. 151–172, doi:10.1007/978-3-319-48628-4_7.

[5] Sol Swords (Accessed: 2023): *ACL2 documentation: Def-bounds*. Available at `https://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2____DEF-BOUNDS`.

A Formalization of Finite Group Theory: Part II

David M. Russinoff
david@russinoff.com

This is the second installment of an exposition of an ACL2 formalization of finite group theory. The first, which was presented at the 2022 ACL2 workshop, covered groups and subgroups, cosets, normal subgroups, and quotient groups, culminating in a proof of Cauchy's Theorem: *If the order of a group G is divisible by a prime p, then G has an element of order p.* This sequel addresses homomorphisms, direct products, and the Fundamental Theorem of Finite Abelian Groups: *Every finite abelian group is isomorphic to the direct product of a list of cyclic p-groups, the orders of which are unique up to permutation.* This theorem is a suitable application of ACL2 because of its extensive reliance on recursion and induction as well as the constructive nature of the factorization. The proof of uniqueness is especially challenging, requiring the formalization of vague intuition that is commonly taken as self-evident.

1 Introduction

In comparison to higher-order logic theorem provers, ACL2 offers a high degree of proof automation at the expense of logical expressiveness. With regard to the formalization of pure mathematics, basic concepts are often difficult to formulate in a first-order logic, but when this obstacle is overcome, proofs are relatively straightforward. This prospect has motivated our pursuit of an ACL2 formalization of abstract algebra, beginning with finite group theory, an area in which substantial progress has already been achieved with other provers, most notably the Coq proof assistant [1]. Our investigation of this subject, which deals with properties of operations defined on sets, must address two limitations of the ACL2 logic: (1) quantification over functions is not provided, and (2) ACL2 data are ordered lists rather than sets. A common solution to these problems is the use of constrained functions. In particular, a natural approach to the formalization of group theory begins with an `encapsulate` form that introduces a set of constrained functions including a predicate representing group membership, a binary group operation, and a unary inverse operator. An alternative scheme based on `defn-sk` was devised by Yuan Yu in his 1990 Nqthm formalization [4], which included a proof of Lagrange's Theorem: *The order of a group is divisible by that of any subgroup.*

Our original submission on this subject to the 2022 ACL2 Workshop [3] was motivated by the observation that any significant progress beyond Lagrange's Theorem would require the facility of proof by induction on the order of a group, which is apparently unavailable through either of the methods mentioned above. That is, a more productive ACL2 formalization of group theory would begin with an explicit predicate that recognizes groups of arbitrary well-defined orders. (This approach necessarily limits the investigation to finite groups.) Thus, the predicate `groupp` defines a group of order n to be an $n \times n$ matrix (a list of n rows of length n) representing the group's operation table, which is stipulated to satisfy the usual group axioms. The first row of the matrix is the list of group elements, the order of which is insignificant except that the first element must be the identity:

```
(defmacro elts (g) `(car ,g))
(defmacro in (x g) `(member-equal ,x (elts ,g)))
(defund e (g) (caar g))
```

A. Coglio, S. Swords (Eds.): 18th ACL2 Workshop (ACL2-2023)
EPTCS 393, 2023, pp. 16–32, doi:10.4204/EPTCS.393.4

The index of an element x of g, (ind x g), is its position in the element list:

```
(defun index (x l)
  (if (consp l)
      (if (equal x (car l))
          0
        (1+ (index x (cdr l))))
    0))
(defmacro ind (x g) `(index ,x (elts ,g)))
```

Thus, the group operation is defined by

```
(defund op (x y g)
  (nth (ind y g)
       (nth (ind x g) g)))
```

The lack of ACL2 support for unordered sets is mitigated by exploiting the notion of an ordered list of elements of a group. Thus, for example, a left coset of a subgroup is defined to be ordered with respect to the larger group, thereby ensuring that intersecting cosets are equal. Note that a subgroup need not be ordered with respect to its parent. For example, a cyclic subgroup has a natural ordering that is generally different from that of the parent group. In most cases, however, we arrange for the ordering to be inherited.

In order to circumvent the cumbersome explicit construction of the defining table for every group of interest, we introduce a `defgroup` macro that generates a parametrized group definition and proofs of the axioms (through functional instantiation) once the user has supplied the element list and terms specifying the binary operation and the inverse operator. This is used, for example, in the definition of the quotient group of a normal subgroup as well as the symmetric groups. A similar `defsubgroup` macro facilitates the definition of parametrized subgroups, e.g., the centralizer of a group element, the center of a group, cyclic subgroups, and intersections of subgroups. As a proof of concept, the culmination of the 2022 submission is an inductive proof of a theorem of Cauchy: *If the order of a group G is divisible by a prime p, then G has an element of order p.* The theory up to this point is embodied in the first four books of the ACL2 directory `books/projects/groups`: `lists`, `groups`, `quotients`, and `cauchy`.

The 2022 paper is a prerequisite for a reading of this sequel, which presents a continuation of the theory comprising three additional books of the `groups` directory. The first of these, `maps`, addresses another class of functions that must be encoded in the logic: group homomorphisms (Section 2), which we represent as association lists. The second, `products`, covers direct products (Section 3). The results of these two books are applied in the third, `abelian`, in a proof of the Fundamental Theorem of Finite Abelian Groups: *Every finite abelian group is isomorphic to the direct product of a list of cyclic p-groups, the orders of which are unique up to permutation.* The proof also applies various number-theoretic results from the book `projects/numbers/euclid`. The proof is in three parts: (1) the factorization of an abelian p-group as a product of cyclic groups (Section 4, (2) the factorization of an arbitrary abelian group as a product of p-groups (Section 5), and (3) the uniqueness of the factorization (Section 6). This theorem is a suitable application of ACL2 because of its extensive reliance on recursion and induction as well as the constructive nature of the factorization. The proof of uniqueness is especially challenging, requiring the formalization of vague intuition that is usually taken as self-evident [2].

2 Homomorphisms

A *homomorphism* is a function `f` from a group `g` to a group `h` that preserves the group operation, i.e., satisfies the identity `(op x y g) = (op (f x) (f y) h)`, and thus relates the algebraic properties

of g and h. Of particular interest is the case of a bijective homomorphism, or an *isomorphism*, which establishes the algebraic equivalence of two groups.

In order to introduce group homomorphisms into our theory, we define a *map* to be an alist with pairwise distinct cars:

```
(defun cons-listp (m)
  (if (consp m) (and (consp (car m)) (cons-listp (cdr m))) (null m)))
(defund domain (m) (strip-cars m))
(defund mapp (m) (and (cons-listp m) (dlistp (domain m))))
```

The function mapply applies a map to an element of its domain:

```
(defund mapply (map x) (cdr (assoc-equal x map)))
```

The macro defmap provides a convenient method of defining maps. The macro call

(defmap *name args domain val*)

defines a family of maps parametrized by *args* with given *domain*, which is assumed to be a dlist. The parameter *val* is the value that the map assigns to an element x of its domain. For example, the following form automates the construction of a composition of two maps:

```
(defmap compose-maps (map2 map1)
  (domain map1)
  (mapply map2 (mapply map1 x)))
```

Evaluation of this form generates two definitions and proves three lemmas:

```
(DEFUN COMPOSE-MAPS-AUX (L MAP2 MAP1)
  (IF (CONSP L)
      (LET ((X (CAR L)))
        (CONS (CONS X (MAPPLY MAP2 (MAPPLY MAP1 X)))
              (COMPOSE-MAPS-AUX (CDR L) MAP2 MAP1)))
    NIL))
(DEFUN COMPOSE-MAPS (MAP2 MAP1)
  (COMPOSE-MAPS-AUX (DOMAIN MAP1) MAP2 MAP1))
(DEFTHM DOMAIN-COMPOSE-MAPS
  (IMPLIES (DLISTP (DOMAIN MAP1))
           (EQUAL (DOMAIN (COMPOSE-MAPS MAP2 MAP1))
                  (DOMAIN MAP1))))
(DEFTHM MAPP-COMPOSE-MAPS
  (IMPLIES (DLISTP (DOMAIN MAP1))
           (MAPP (COMPOSE-MAPS MAP2 MAP1))))
(DEFTHM COMPOSE-MAPS-VAL
  (IMPLIES (MEMBER-EQUAL X (DOMAIN MAP1))
           (EQUAL (MAPPLY (COMPOSE-MAPS MAP2 MAP1) X)
                  (MAPPLY MAP2 (MAPPLY MAP1 X)))))
```

A *homomorphism* from a group g to a group h is a map that satisfies the following predicate:

```
(defund homomorphismp (map g h)
  (and (groupp g)
       (groupp h)
       (mapp map)
       (sublistp (elts g) (domain map))
       (equal (mapply map (e g)) (e h))
       (not (codomain-cex map g h))
       (not (homomorphism-cex map g h))))
```

The functions codomain-cex and homomorphism-cex search for counterexamples of these two properties:

```
(implies (in x g)
         (in (mapply map x) h))
(implies (and (in x g) (in y g))
         (equal (mapply map (op x y g))
                (op (mapply map x) (mapply map y) h)))
```

Once these two implications have been proved to hold for a proposed homomorphism, the corresponding conjuncts of the definition are readily established. In this manner, it is easily shown, for example, that a composition of homomorphisms is a homomorphism:

```
(defthm homomorphismp-compose-maps
  (implies (and (homomorphismp map1 g h) (homomorphismp map2 h k))
           (homomorphismp (compose-maps map2 map1) g k)))
```

The *image* of a homomorphism map from g to h is a subgroup of h. Its element list is defined using the function insert, which ensures that it is ordered with respect to h:

```
(defun ielts-aux (map l h)
  (if (consp l)
      (insert (mapply map (car l))
              (ielts-aux map (cdr l) h)
              h)
    ()))
(defund ielts (map g h)
  (ielts-aux map (elts g) h))
(defthm ordp-ielts
  (implies (homomorphismp map g h)
           (ordp (ielts map g h) h)))
```

The group (image g h) is automatically defined by defsubgroup once the usual prerequisite lemmas have been proved:

```
(defthm dlistp-ielts
  (implies (homomorphismp map g h)
           (dlistp (ielts map g h))))
(defthm sublistp-ielts
  (implies (homomorphismp map g h)
           (sublistp (ielts map g h) (elts h))))
(defthm consp-ielts
  (implies (groupp g)
           (consp (ielts map g h))))
(defthm ielts-closed
  (implies (and (homomorphismp map g h)
                (member-equal x (ielts map g h))
                (member-equal y (ielts map g h)))
           (member-equal (op x y h) (ielts map g h))))
(defthm ielts-inverse
  (implies (and (homomorphismp map g h)
                (member-equal x (ielts map g h)))
           (member-equal (inv x h) (ielts map g h))))
```

The arguments of defsubgroup are the subgroup parameters, the parent group, the constraint that must be satisfied by the parameters, and the element list:

```
(defsubgroup image (map g) h
  (homomorphismp map g h)
  (ielts map g h))
```

The *kernel* of `map` is a subgroup of `g` consisting of those elements that are mapped to the identity element of `h`:

```
(defun kelts-aux (map l h)
  (if (consp l)
      (if (equal (mapply map (car l)) (e h))
          (cons (car l) (kelts-aux map (cdr l) h))
        (kelts-aux map (cdr l) h))
  ()))
(defund kelts (map g h)
  (kelts-aux map (elts g) h))
```

Once again, we invoke `defsubgroup` after establishing its prerequisite lemmas:

```
(defsubgroup kernel (map h) g
  (homomorphismp map g h)
  (kelts map g h))
```

The usual classes of homomorphisms are defined in terms of the image and the kernel:

```
(defund epimorphismp (map g h)
  (and (homomorphismp map g h)
       (equal (image map g h) h)))
(defund endomorphismp (map g h)
  (and (homomorphismp map g h)
       (equal (kernel map h g) (trivial-subgroup g))))
(defund isomorphismp (map g h)
  (and (epimorphismp map g h) (endomorphismp map g h)))
```

The inverse of an isomorphism is defined by `defmap`:

```
(defun preimage-aux (x map l)
  (if (consp l)
      (if (equal x (mapply map (car l)))
          (car l)
        (preimage-aux x map (cdr l)))
    ()))
(defund preimage (x map g)
  (preimage-aux x map (elts g)))
(defmap inv-isomorphism (map g h) (elts h) (preimage x map g))
(defthmd isomorphismp-inv
  (implies (isomorphismp map g h)
           (isomorphismp (inv-isomorphism map g h) h g)))
```

We shall also require the following important property of isomorphisms:

```
(defthm isomorphismp-compose-maps
  (implies (and (isomorphismp map1 g h) (isomorphismp map2 h k))
                (isomorphismp (compose-maps map2 map1) g k)))
```

3 Direct Products

Direct products provide a means of constructing complex groups from simpler ones. Given a non-null proper list of groups `l`, we shall define the group `(direct-product l)`. Its element list is the Cartesian product `(group-tuples l)`, defined as follows:

```
(defun conses (x l)
  (if (consp l)
      (cons (cons x (car l)) (conses x (cdr l)))
    ()))
(defun group-tuples-aux (l m)
  (if (consp l)
      (append (conses (car l) m)
              (group-tuples-aux (cdr l) m))
    ()))
(defun group-tuples (l)
  (if (consp l)
      (group-tuples-aux (elts (car l)) (group-tuples (cdr l)))
    (list ())))
```

It is easily shown that the length of `(group-tuples l)` is the product of the orders of the members of `l`. If `x` is a member of this list, then `x` is a list of the same length as `l`, and each member of `x` is a group element of the corresponding member of `l`. The `car` of `(group-tuples l)`, which will be the identity element of the direct product, is the list defined as follows:

```
(defun e-list (l)
  (if (consp l)
      (cons (e (car l)) (e-list (cdr l)))
    ()))
```

The group operation and inverse operator are defined recursively:

```
(defun dp-op (x y l)
  (if (consp l)
      (cons (op (car x) (car y) (car l))
            (dp-op (cdr x) (cdr y) (cdr l)))
    ()))
(defun dp-inv (x l)
  (if (consp l)
      (cons (inv (car x) (car l))
            (dp-inv (cdr x) (cdr l)))
    ()))
```

Once the requisite group properties are proven, we invoke `defgroup` to construct the direct product:

```
(defgroup direct-product (l)
  (and (group-list-p l) (consp l)) ;parameter constraint
  (group-tuples l)                 ;element list
  (dp-op x y l)                    ;group operation
  (dp-inv x gl))                   ;inverse operator
```

The ordering of the elements of the direct product is given by the following, which is useful in proving that a given subgroup is ordered:

```
(defthmd ind-dp-compare
  (implies (and (group-list-p l)
                (consp l)
                (in x (direct-product l))
                (in y (direct-product l)))
           (iff (< (ind x (direct-product l))
                   (ind y (direct-product l)))
                (or (< (ind (car x) (car l))
                       (ind (car y) (car l)))
```

```
                (and (consp (cdr l))
                     (equal (car x) (car y))
                     (< (ind (cdr x) (direct-product (cdr l)))
                        (ind (cdr y) (direct-product (cdr l))))))))
```

A construction related to the direct product is the list of products of elements of two subgroups h and k of a group g. Note that the definition ensures that this list is ordered with respect to g:

```
(defun products-aux (l g)
  (if (consp l)
      (insert (op (caar l) (cadar l) g) (products-aux (cdr l) g) g)
    ()))
(defund products (h k g)
  (products-aux (group-tuples (list h k)) g))
```

While (products h k g) does not in general form a subgroup of g, it does when either h or k is normal:

```
(defsubgroup product-group (h k) g
  (and (subgroupp h g)
       (subgroupp k g)
       (or (normalp h g) (normalp k g)))
  (products h k g))
```

When h and k are both normal in g, so is (product-group (h k g).

We have the following formula for the length of (products h k g):

```
(defthmd len-products
  (implies (and (subgroupp h g)
                (subgroupp k g))
           (equal (len (products h k g))
                  (/ (* (order h) (order k))
                     (order (group-intersection h k g))))))
```

The derivation of this formula is based on the following function, which converts a list l of members of (lcosets (group-intersection h k g) h) to a list of members of (lcosets k g) by replacing each member c of l with (lcoset (car c) k g):

```
(defun lift-cosets-aux (l k g)
  (if (consp l)
      (cons (lcoset (caar l) k g)
            (lift-cosets-aux (cdr l) k g))
    ()))
```

We apply lift-cosets-aux to the full list (lcosets (group-intersection h k g) g):

```
(defund lift-cosets (h k g)
  (lift-cosets-aux (lcosets (group-intersection h k g) h) k g))
```

The result is a list of distinct elements of (lcosets k g), and therefore, appending them yields a dlist:

```
(defthm dlistp-append-list-lift-cosets
  (implies (and (subgroupp h g) (subgroupp k g))
           (dlistp (append-list (lift-cosets h k g)))))
```

The length of (lift-cosets h k g) is that of (lcosets (group-intersection h k g) h), which is the quotient

```
(/ (order h) (order (intersect-groups h k g)))
```

Since the length of each member of the list is `(order k)`, we have the following expression for the length of the appended list:

```
(defthm len-append-list-lift-cosets
  (implies (and (subgroupp h g) (subgroupp k g))
           (equal (len (append-list (lift-cosets h k g)))
                  (/ (* (order h) (order k))
                     (order (group-intersection h k g))))))
```

It is easy to show that each member of `(lift-cosets h k g)` is a sublist of `(products h k g)`. On the other hand, if `x` is an element of `(products h k g)`, then `x = (op a b g)`, where `a` is in `h` and `b` is in `k`. Some member of (lcosets (group-intersection h k g) h) contains `a`, as does the corresponding member of `(lift-cosets h k g)`, which therefore contains `x`. Thus, `(products h k g)` is a sublist of `(append-list (lift-cosets h k g))`. Since both lists are dlists and each is a sublist of the other, they have the same length, and the formula `len-products` follows from `len-append-list-lift-cosets`.

The product of a list of subgroups is defined recursively:

```
(defun product-group-list (l g)
  (if (consp l)
      (product-group (car l) (product-group-list (cdr l) g) g)
    (trivial-subgroup g)))
```

By induction, if each group in `l` is normal in g, then so is `(product-group-list l g)`. If `l` satisfies the following predicate, then that subgroup is isomorphic to `(direct-product l)`:

```
(defun internal-direct-product-p (l g)
  (if (consp l)
      (and (internal-direct-product-p (cdr l) g)
           (normalp (car l) g)
           (equal (group-intersection (car l) (product-group-list (cdr l) g) g)
                  (trivial-subgroup g)))
    (null l)))
```

Moreover, if that subgroup has the same order as g, then since it inherits the ordering of g, the two groups are equal. The isomorphism is conveniently constructed by `defmap`:

```
(defun product-list-val (x g)
  (if (consp x)
      (if (consp (cdr x))
          (op (car x) (product-list-val (cdr x) g) g)
        (car x))
    ()))
(defmap product-list-map (l g)
  (group-tuples l)
  (product-list-val x g))
(defthmd isomorphismp-dp-idp
  (implies (and (groupp g)
                (consp l)
                (internal-direct-product-p l g)
                (= (product-orders l) (order g)))
           (isomorphismp (product-list-map l g)
                         (direct-product l)
                         g)))
```

Note also that the product of two non-intersecting internal direct products is an internal direct product:

```
(defthmd internal-direct-product-append
  (implies (and (internal-direct-product-p l g)
                (internal-direct-product-p m g)
                (equal (group-intersection (product-group-list l g)
                                           (product-group-list m g)
                                           g)
                       (trivial-subgroup g)))
           (internal-direct-product-p (append l m) g)))
```

4 Factorization of p-Groups

A group is *abelian* if its operation is commutative. The notion of an abelian group may be viewed as an abstraction of the familiar numerical groups: the additive groups of integers, modular integers, rationals, and reals, and the multiplicative groups of the non-zero rationals and reals. Finite abelian groups admit a particularly simple classification as direct products of cyclic groups. We begin with the case of an abelian `p-group`, the order of which is a power of a prime p:

```
(defund p-groupp (g p)
  (and (primep p) (groupp g) (powerp (order g) p)))
```

In this section, we shall prove that every abelian p-group is an internal direct product of cyclic subgroups. This will follow by induction once we show that if such a group is not cyclic, then it is an internal direct product of two non-trivial subgroups. The proof of this result is also inductive, based on the notion of "lifting" a subgroup of a quotient group. If `n` is a normal subgroup of `g` and `h` is a subgroup of `(quotient g n)`, then `(lift h n g)` is the subgroup of `g` formed by appending the cosets that belong to `h`. In the book `groups/quotients`, we prove the following two lemmas:

```
(defthmd lift-subgroup
  (implies (and (normalp n g) (subgroupp h (quotient g n)))
           (subgroupp n (lift h n g))))

(defthmd lift-order
  (implies (and (normalp n g) (subgroupp h (quotient g n)))
           (equal (order (lift h n g)) (* (order h) (order n)))))
```

Assume `(p-groupp g p)` and `(abelianp g)`. Our objective is to construct subgroups `g1` and `g2` of `g` that satisfy the following predicate:

```
(defund desired-properties (g g1 g2)
  (and (subgroupp g1 g)
       (cyclicp g1)
       (subgroupp g2 g)
       (equal (* (order g1) (order g2)) (order g))
       (equal (group-intersection g1 g2 g) (trivial-subgroup g))))
```

The construction begins with the selection of an element `a` of maximal ord in `g`, as computed by the function `max-ord`:

```
(defun max-ord-aux (g n)
  (if (zp n)
      1
    (if (elt-of-ord n g)
        n
      (max-ord-aux g (1- n)))))
(defund max-ord (g)
  (max-ord-aux g (order g)))
```

For convenience, we collect these hypotheses in another predicate:

```
(defund phyp (a p g)
  (and (p-groupp g p)
       (abelianp g)
       (not (cyclicp g))
       (in a g)
       (equal (ord a g) (max-ord g))))
```

The first of the two subgroups is the cyclic group generated by a:

```
(defund g1 (a g) (cyclic a g))
```

(We shall abbreviate the term (g1 a g) as g1; related terms defined below will be similarly abbreviated.) By cauchy, some coset in (quotient g g1) has order p. We define x$ to be a member of that coset:

```
(defund x$ (a p g) (car (elt-of-ord p (quotient g (g1 a g)))))
```

Thus, x$ is not in g1 but (power x$ p g) is in g1. This implies (power x$ p g) is a member of (powers a g), and hence (power x$ p g) = (power a i$ g), where i$ is defined by

```
(defund i$ (a p g) (index (power (x$ a p g) p g) (powers a g)))
```

It follows that i$ is divisible by p, for otherwise (power a i$ g) has the same order as a, implying that x$ has order greater than (max-ord g). Consider the cyclic subgroup c$ of g defined as follows:

```
(defund j$ (a p g) (/ (i$ a p g) p))

(defund y$ (a p g)
  (op (power (inv a g) (j$ a p g) g)
      (x$ a p g)
      g))

(defun c$ (a p g) (cyclic (y$ a p g) g))
```

Note that y$ is not in g1, but

```
(power y$ p g) = (op (power (power (inv a g) j$ g) p g) (power x$ p g) g)
               = (op (power (inv a g) (* j$ p) g) (power x$ p g) g)
               = (op (power (inv a g) i$ g) (power x$ p g) g)
               = (op (inv (power a i$ g) g) (power x$ p g) g)
               = (op (inv (power x$ p g) g) (power x$ p g) g)
               = (e g)
```

Thus, (order c$) = (ord y$ g) = p and g1 and c$ intersect trivially. Let g* and a* be defined as follows:

```
(defun g* (a p g)
  (quotient g (c$ a p g)))
(defun a* (a p g)
  (lcoset a (c$ a p g) g))
```

Then (ord a* g*) = (max-ord g), for otherwise a power of a less than (max-ord g) would be in c$ and therefore equal to (e g). Thus, a* has maximal ord in g$. If g* is cyclic, then its order is (max-ord g), which implies

```
(order g) = (* (order g*) p} = (* (order g1) (order c$))
```

and we may define g2 = c$. Otherwise we proceed by induction on (order g, substituting g* and a* for g and a. Let g1* = (cyclic a* g*). By inductive hypothesis, g* is the internal direct product of g1* and some g2*. We define g2 to be (lift g2* c$ g). This yields the following recursive definition:

```
(defund g2 (a p g)
  (declare (xargs :measure (order g)))
  (if (phyp a p g)
      (if (cyclicp (g* a p g))
          (c$ a p g)
        (lift (g2 (a* a p g) p (g* a p g))
              (c$ a p g)
              g))
    ()))
```

We need only show that `(desired-properties g g1 g2)` follows from `(desired-properties g* g1* g2*)`. We may assume that `g1` is not cyclic. We have `(order g1) = (max-ord g) = (order g1*)`, and by `lift-order`, `(order g2) = (* p (order g2*)`. Thus,

```
(* (order g1) (order g2)) = (* p (order g1*) (order g2*))
                           = (* p (order g*))
                           = (order g).
```

Finally, suppose `r` is in both `g1` and `g2`. Then `(lcoset r c$ g)` is in both `g1*` and `g2*`, which implies `r` is in `c$`. But then `r` is in both `g1` and `c$`, which implies `r = (e g)`. This completes the induction, and we have

```
(defthmd factor-p-group
  (implies (phyp a p g)
           (desired-properties g (g1 a g) (g2 a p g))))
```

This result provides the basis of the factorization of `g`. The goal is to show that `g` is the internal direct product of a list of subgroups characterized as follows:

```
(defund cyclic-p-group-p (g)
  (and (cyclicp g)
       (> (order g) 1)
       (p-groupp g (least-prime-divisor (order g)))))
(defun cyclic-p-group-list-p (l)
  (if (consp l)
      (and (cyclic-p-group-p (car l)) (cyclic-p-group-list-p (cdr l)))
    (null l)))
```

The list is constructed recursively:

```
(defun cyclic-p-subgroup-list (p g)
  (declare (xargs :measure (order g)))
  (if (and (p-groupp g p) (abelianp g) (> (order g) 1))
      (if (cyclicp g)
          (list g)
        (let ((a (elt-of-ord (max-ord g) g)))
          (cons (g1 a g) (cyclic-p-subgroup-list p (g2 a p g)))))
    ()))
```

The desired result follows from `factor-p-group` by induction on `(order g)`:

```
(defthmd p-group-factorization
  (implies (and (p-groupp g p) (abelianp g) (> (order g) 1))
           (let ((l (cyclic-p-subgroup-list p g)))
             (and (consp l)
                  (cyclic-p-group-list-p l)
                  (internal-direct-product-p l g)
                  (equal (order g) (product-orders l))))))
```

5 Factorization of Abelian Groups

We shall prove constructively that every finite abelian group is isomorphic to a direct product of cyclic p-groups. The proof is again inductive, based on p-group-factorization and the following lemma: *If* (order g) *is the product of relatively prime integers* m *and* n, *then* g *is the internal direct product of two subgroups of orders* m *and* n. To derive this result, we define the ordered list of all elements of g with order dividing m:

```
(defun elts-of-ord-dividing-aux (m l g)
  (if (consp l)
      (if (divides (ord (car l) g) m)
          (cons (car l) (elts-of-ord-dividing-aux m (cdr l) g))
        (elts-of-ord-dividing-aux m (cdr l) g))
    ()))
(defund elts-of-ord-dividing (m g)
  (elts-of-ord-dividing-aux m (elts g) g))
```

If g is abelian, then this list forms a subgroup of g:

```
(defsubgroup subgroup-ord-dividing (m) g
  (and (abelianp g) (posp m))
  (elts-of-ord-dividing m g))
```

Let h = (subgroup-ord-dividing m g) and k = (subgroup-ord-dividing n g). If x is in both h and k, then (ord x) divides both m and n, and by divides-gcd of the book euclid, (ord x) divides (gcd m n) = 1, which implies x = (e g). Thus, h and k intersect trivially:

```
(defthmd rel-prime-factors-intersection
  (implies (and (groupp g) (abelianp g)
                (posp m) (posp n) (= (gcd m n) 1))
           (let ((h (subgroup-ord-dividing m g)) (k (subgroup-ord-dividing n g)))
             (equal (group-intersection h k g) (trivial-subgroup g)))))
```

By gcd-linear-combination (book euclid), since (gcd m n) = 1, there exist r and s such that (+ (* r n) (* s m)) = 1. Let x be in g. Then

```
x = (power x (+ (* r n) (* s m)) g) = (op (power x (* r n) g) (power x (* s m) g) g).
```

Since

```
(power (power x (* r n) g) m g) = (power (power x (* m n) g) r g) = (e g),
```

(power x (* r n) g) is in m, and similarly, (power x (* s m) g) is in k. Thus, by len-products,

```
(* m n) = (order g) = (len (products h k g)) = (* (order h) (order k)).
```

If p is a prime dividing (order h), then by cauchy, h has an element of order p, and therefore p divides m, which implies p does not divide n. By lagrange and divides-product-divides-factor (of the book euclid), (order h) divides m, and therefore (<= (order h) m). Similarly, (<= (order k) n). Since (* m n) = (* (order h) (order k)), both equalities must hold:

```
(defthmd rel-prime-factors-orders
  (implies (and (groupp g) (abelianp g)
                (posp m) (posp n) (= (gcd m n) 1)
                (= (order g) (* m n)))
           (let ((h (subgroup-ord-dividing m g)) (k (subgroup-ord-dividing n g)))
             (and (equal (order h) m) (equal (order k) n)))))
```

Let p be the least prime divisor of (order g). Let m be the maximum power of p that divides (order g) and let n = (/ (order g) m). Then m and n are relatively prime. We define a list of subgroups of g recursively, using cyclic-p-subgroup-list:

```
(defun cyclic-subgroup-list (g)
  (declare (xargs :measure (order g) ))
  (if (and (groupp g) (abelianp g))
      (if (= (order g) 1)
          ()
        (let* ((p (least-prime-divisor (order g)))
               (m (max-power-dividing p (order g)))
               (n (/ (order g) m))
               (h (subgroup-ord-dividing m g))
               (k (subgroup-ord-dividing n g)))
          (append (cyclic-p-subgroup-list p h)
                  (cyclic-subgroup-list k))))
    ()))
```

The following is proved by induction, combining `rel-prime-factors-intersection` and `rel-prime-factors-orders` with `internal-direct-product-append` (Section 3) and `p-group-factorization` (Section 4):

```
(defthmd idp-cyclic-subgroup-list
  (implies (and (groupp g) (abelianp g) (> (order g) 1))
           (let ((l (cyclic-subgroup-list g)))
             (and (cyclic-p-group-list-p l)
                  (internal-direct-product-p l g)
                  (equal (product-orders l) (order g))))))
```

Finally, we invoke isomorphismp-dp-idp (Section 3):

```
(defthmd abelian-factorization
  (implies (and (groupp g) (abelianp g) (> (order g) 1))
           (let ((l (cyclic-subgroup-list g)))
             (and (cyclic-p-group-list-p l)
                  (isomorphismp (product-list-map l g) (direct-product l) g)))))
```

6 Uniqueness of the Factorization

We define the list of orders of a list of groups:

```
(defun orders (l)
  (if (consp l)
      (cons (order (car l)) (orders (cdr l)))
    ()))
```

Our objective is to show that if the direct products of two lists of cyclic p-groups `l` and `m` are isomorphic, then `(orders l)` and `(orders m)` satisfy the following predicate, which is defined in the book `lists`:

```
(defun permutationp (l m)
  (if (consp l)
      (and (member-equal (car l) m)
           (permutationp (cdr l) (remove1-equal (car l) m)))
    (endp m)))
```

We shall make use of an equuivalent formulation of `permutationp`, based on a function that counts the number of occurrences of an object in a list:

```
(defun hits (x l)
  (if (consp l)
      (if (equal x (car l))
```

```
          (1+ (hits x (cdr l)))
        (hits x (cdr l)))
    0))
```

It is evident that `(permutationp l m)` holds iff `(hits x l) = (hits x m)` for all `x`. The formalization of this claim requires a function that searches for a counterexample:

```
(defun hits-diff-aux (test l m)
  (if (consp test)
      (if (equal (hits (car test) l) (hits (car test) m))
          (hits-diff-aux (cdr test) l m)
        (list (car test)))
    ()))
(defund hits-diff (l m) (hits-diff-aux (append l m) l m))
(defthmd hits-diff-perm (iff (permutationp l m) (not (hits-diff l m))))
```

The uniqueness proof is also based on the notion of a power of an abelian group. We define the list of `n`th powers of the elements of `g`:

```
(defun power-list-aux (l n g)
  (if (consp l)
      (insert (power (car l) n g)
              (power-list-aux (cdr l) n g)
              g)
    ()))
(defun power-list (n g)
  (power-list-aux (elts g) n g))
```

If `g` is abelian, then this list forms a subgroup of `g`:

```
(defsubgroup group-power (n) g
  (and (posp n) (groupp g) (abelianp g))
  (power-list n g))
```

If two abelian groups are isomorphic, then so are their `n`th powers:

```
(defthmd isomorphismp-power
  (implies (and (isomorphismp map g h) (abelianp g) (posp n))
           (isomorphismp map (group-power n g) (group-power n h))))
```

The `n`th power of a direct product of abelian groups is the direct product of the `n`th powers. The proof is more challenging than expected, as it requires showing not only that each element list is a sublist of the other, but also that both lists are ordered with respect to `(direct-product l)`, according to the lemma `ind-dp-compare` (Section 3):

```
(defun group-power-list (n l)
  (if (consp l)
      (cons (group-power n (car l))
            (group-power-list n (cdr l)))
    ()))
(defthmd group-power-dp
  (implies (and (posp n) (consp l) (abelian-list-p l))
           (equal (group-power n (direct-product l))
                  (direct-product (group-power-list n l)))))
```

The `n`th power of a cyclic group is cyclic. For prime `p`, the order of `(group-power p g)` depends on whether `p` divides the order of `g`:

```
(defun reduce-order (n p)
  (if (divides p n) (/ n p) n))
(defthmd prime-power-cyclic
  (implies (and (cyclicp g) (primep p))
           (and (cyclicp (group-power p g))
                (equal (order (group-power p g))
                       (reduce-order (order g) p)))))
```

The list of orders of `(group-power-list p l)`:

```
(defun reduce-orders (orders p)
  (if (consp orders)
      (cons (reduce-order (car orders) p) (reduce-orders (cdr orders) p))
    ()))
(defthm order-group-power-list
  (implies (and (primep p) (cyclic-p-group-list-p l))
           (equal (orders (group-power-list p l))
                  (reduce-orders (orders l) p))))
```

It is a simple matter to identify a prime that divides at least one of the orders:

```
(defund first-prime (l) (least-prime-divisor (order (car l))))
```

Let `p = (first-prime l)`. We would like to use an induction scheme that replaces `l` and `m` with `(group-powers p l)` and `(group-powers p m)`, but in order to ensure that these lists inherit the properties of `l` and `m`, we must delete any occurrences of trivial groups:

```
(defun delete-trivial (l)
  (if (consp l)
      (if (= (order (car l)) 1)
          (delete-trivial (cdr l))
        (cons (car l) (delete-trivial (cdr l))))
    ()))
(defund reduce-cyclic (l p) (delete-trivial (group-power-list p l)))
```

Let `l' = (reduce-cyclic l p)` and `m' = (reduce-cyclic m p)`. The properties of l and m are inherited by `l'` and `m'`:

```
(defthmd reduce-cyclic-p-group-list
  (implies (and (primep p) (cyclic-p-group-list-p l))
           (cyclic-p-group-list-p (reduce-cyclic l p))))
```

We would like to show that if `(orders l')` is a permutation of `(orders m')`, then the same is true of `l` and `m`. By `hits-diff-perm`, it suffices to show that for all `x`, `(hits x (orders l)) = (hits x (orders m))`. It may be proved as a consequence of `order-group-power-list` that this holds for all `x` other than `p`. But note that `(orders l)` and `(orders m)` have the same product:

```
(product-orders l) = (order (direct-product l))
                   = (order (direct-product l))
                   = (product-orders m).
```

It follows that the equation holds for `x = p` as well. Thus, we have

```
(defthmd permutationp-orders
  (implies (and (consp l)
                (consp m)
                (cyclic-p-group-list-p l)
                (cyclic-p-group-list-p m)
                (primep p)
                (isomorphismp map (direct-product l) (direct-product m))
                (permutationp (orders (reduce-cyclic l p))
                              (orders (reduce-cyclic m p))))
           (permutationp (orders l) (orders m))))
```

The base case of the induction is `(or (null l') (null m'))`. If `l' = nil`, then every element of `l` must be a group of order `p`, which then every non-trivial element of `(direct-product l)` has order `p`. But then the same must be true of `(direct-product m)` and consequently m' = nil. Therefore, if either l' or m' is null, then

```
(orders (reduce-cyclic l p)) = (orders (reduce-cyclic l p)) = nil.
```

In particular, `(permutationp (orders l') (orders m'))` and we have the following consequence of `permutationp-orders`:

```
(defthmd null-reduce-cyclic-case
  (implies (and (consp l)
                (consp m)
                (cyclic-p-group-list-p l)
                (cyclic-p-group-list-p m)
                (primep p)
                (or (null (reduce-cyclic l p))
                    (null (reduce-cyclic m p)))
                (isomorphismp map (direct-product l) (direct-product m)))
           (permutationp (orders l) (orders m))))
```

In the remaining inductive case, we need only show that if `(direct-product l)` and `(direct-product m)` are isomorphic, then so are `(direct-product l')` and `(direct-product m')`. We begin by constructing an isomorphism between `(direct-product (group-power-list p l))` and `(direct-product l')`:

```
(defun delete-trivial-elt (x l)
  (if (consp x)
      (if (= (order (car l)) 1)
          (delete-trivial-elt (cdr x) (cdr l))
        (cons (car x) (delete-trivial-elt (cdr x) (cdr l))))
    ()))
(defmap delete-trivial-iso (l)
  (group-tuples l)
  (delete-trivial-elt x l))
(defthmd isomorphismp-delete-trivial
  (implies (and (group-list-p l) (consp (delete-trivial l)))
           (isomorphismp (delete-trivial-iso l)
                         (direct-product l)
                         (direct-product (delete-trivial l)))))
```

Now suppose `map` is an isomorphism from `(direct-product l)` to `(direct-product m)`. By `isomorphismp-power`,

```
(isomorphismp map (group-power p (direct-product l))
                  (group-power p (direct-product m))),
```

and by `group-power-dp`,

```
(isomorphismp map (direct-product (group-power-list p l))
                  (direct-product (group-power-list p m))).
```

Thus, the desired isomorphism is constructed as a composition of three isomorphisms:

```
(defund reduce-cyclic-iso (map l m p)
  (compose-maps
    (delete-trivial-iso (group-power-list p m))
    (compose-maps
      map
      (inv-isomorphism (delete-trivial-iso (group-power-list p l))
                       (direct-product (group-power-list p l))
                       (direct-product (reduce-cyclic l p))))))
```

We apply `isomorphismp-delete-trivial`, `isomorphismp-inv`, and `isomorphismp-compose-maps` to conclude that `reduce-cyclic-iso` is an isomorphism:

```
(defthmd isomorphismp-reduce-cyclic
  (implies (and (consp l)
                (consp m)
                (primep p)
                (cyclic-p-group-list-p l)
                (cyclic-p-group-list-p m)
                (consp (reduce-cyclic l p))
                (consp (reduce-cyclic m p))
                (isomorphismp map (direct-product l) (direct-product m)))
           (isomorphismp (reduce-cyclic-iso map l m p)
                         (direct-product (reduce-cyclic l p))
                         (direct-product (reduce-cyclic m p)))))
```

Our theorem follows from `null-reduce-cyclic-case`, `permutationp-orders`, and `isomorphismp-reduce-cyclic` by induction:

```
(defthmd abelian-factorization-unique
  (implies (and (consp l)
                (consp m)
                (cyclic-p-group-list-p l)
                (cyclic-p-group-list-p m)
                (isomorphismp map (direct-product l) (direct-product m)))
           (permutationp (orders l) (orders m))))
```

References

[1] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi & Laurent Théry (2013): *A Machine-Checked Proof of the Odd Order Theorem*. In: *International Conference on Interactive Theorem Proving*, pp. 163–179, doi:10.1017/S0960129511000132.

[2] Joseph Rotman (1965): *The Theory of Groups: An Introduction*. Allyn and Bacon.

[3] David M. Russinoff (2022): *A Formalization of Finite Froup Theory*. In: *ACL2 2022: 17th International Workshop on the ACL2 Theorem Prover and its Applications*, Austin, Texas, doi:10.4204/EPTCS.359.10.

[4] Yuan Yu (1990): *Computer Proofs in Group Theory*. *Journal of Automated Reasoning* 6(3), doi:10.1007/BF00244488.

A Formalization of Finite Group Theory: Part III

David M. Russinoff
david@russinoff.com

This is the third and final installment of an exposition of an ACL2 formalization of finite group theory. Part I covers groups and subgroups, cosets, normal subgroups, and quotient groups. Part II extends the theory in the developmnent of group homomorphisms and direct products, which are applied in a proof of the Fundamental Theorem of Finite Abelian Groups. The central topics of the present paper are the symmetric groups and the Sylow theorems, which pertain to subgroups of prime power order. Since these theorems are based on the conjugation of subgroups, an example of a group action on a set, their presentation is preceded by a comprehensive treatment of group actions. Our final result is mainly an application of the Sylow theorems: after showing that the alternating group of order 60 is simple (i.e., has no proper normal subgroup), we prove that no group of non-prime order less than 60 is simple. The combined content of the `groups` directory is a close approximation to that of an advanced undergraduate course taught by the author in 1976.

1 Introduction

This is the third and final installment of an exposition of an ACL2 formalization of finite group theory. Part I [1], which was presented at ACL2 2022, covers groups and subgroups, cosets, normal subgroups, and quotient groups. Part II [2], a companion paper in this workshop, extends the theory in the developmnent of group homomorphisms and direct products, which are applied in a proof of the Fundamental Theorem of Finite Abelian Groups. The present paper is an account of four books of the directory `projects/groups`—`symmetric`, `actions`, `sylow`, and `simple`—that have been appended to the seven books described in Parts I and II. Part I is a prerequisite for a reading of this paper. Parts II and III are largely independent, aside from several explicit references to Part II contained herein.

The central topics covered here are the symmetric groups `(sym n)` (Section 2), consisting of the permutations of the list `(0 1 2 ... n-1)`, and the Sylow theorems (Section 4), which pertain to subgroups of prime power order. Since these theorems are based on the conjugation of subgroups, an example of a group action on a set, their presentation is preceded by a comprehensive treatment of group actions (Section 3). Our final result (Section 5) is mainly an application of the Sylow theorems: after showing that the alternating group `(alt 5)`, of order 60, is simple (i.e., has no proper normal subgroup), we prove that no group of non-prime order less than 60 is simple.

2 Symmetric Groups

A *symmetric group* is the group of permutations of a given set under the operation of functional composition. The study of these groups has important applications in diverse areas of mathematics and physics, such as combinatorics, Galois theory, and quantum mechanics. They also provide a wide range of examples in group theory. Since the elements of the underlying set are irrelevant to the group structure, it is common to focus on the permutations of an initial segment of the positive integers, $\{1,2,\ldots,n\}$. In our ACL2 formalization, it is more natural to consider the list `(ninit n)` = `(0 1 ... n-1)` of the first `n` natural numbers.

A. Coglio, S. Swords (Eds.): 18th ACL2 Workshop (ACL2-2023)
EPTCS 393, 2023, pp. 33–49, doi:10.4204/EPTCS.393.5

2.1 Definition of (sym n)

In [2, Sec. 6], we define a permutation of an arbitrary list:

```
(defun permutationp (l m)
  (if (consp l)
      (and (member-equal (car l) m)
           (permutationp (cdr l) (remove1-equal (car l) m)))
    (endp m)))
```

In the special case of a list of distinct members, we have an equivalent formulation:

```
(defund permp (l m)
  (and (dlistp l) (dlistp m) (sublistp l m) (sublistp m l)))
(defthmd permp-permutationp
  (implies (and (dlistp l) (dlistp m))
           (iff (permutationp l m) (permp l m))))
```

The function perms recursively constructs a list of all permutations of a dlist. For a positive integer n, the element list of the symmetric group (sym n) is (slist n), the list of permutations of (ninit n):

```
(defund slist (n) (perms (ninit n)))
```

A permutation x in (slist n) may be viewed as a bijection of (ninit n), which maps an integer k to (nth k x). The group operation is functional composition:

```
(defun comp-perm-aux (x y l)
  (if (consp l)
      (cons (nth (nth (car l) y) x)
            (comp-perm-aux x y (cdr l)))
    ()))
(defund comp-perm (x y n)
  (comp-perm-aux x y (ninit n)))
```

The behavior of a product of permutations x and y is characterized by the following:

```
(defthm nth-comp-perm
  (implies (and (posp n) (natp k) (< k n))
           (equal (nth k (comp-perm x y n)) (nth (nth k y) x))))
```

More generally, we define the product of a list of permutations:

```
(defun comp-perm-list (l n)
  (if (consp l)
      (comp-perm (car l) (comp-perm-list (cdr l) n) n)
    (ninit n)))
```

The inverse operator is defined using the function index [2, Sec. 1], which gives the location of a member of a list:

```
(defun inv-perm-aux (x l)
  (if (consp l)
      (cons (index (car l) x) (inv-perm-aux x (cdr l)))
    ()))
(defund inv-perm (x n)
  (inv-perm-aux x (ninit n)))
```

It is easily shown that (slist n) is a dlist and that (car (slist n)) = (ninit n) is a left identity. After establishing closure, associativity, and the inverse property, we invoke the defgroup macro to construct the group:

```
(defgroup sym (n)
  (posp n)           ;parameter constraint
  (slist n)          ;element list
  (comp-perm x y n)  ;group operation
  (inv-perm x n))    'inverse operator
```

The length of `(slist n)` is easily computed:

```
(defthmd order-sym (implies (posp n) (equal (order (sym n)) (fact n))))
```

2.2 Transpositions

A *transposition* is a permutation in `(sym n)` that interchanges two indices and leaves all others fixed. The function `transpose` constructs the transposition of given indices `i` and `j`:

```
(defun transpose-aux (i j l)
  (if (consp l)
      (if (equal (car l) i)
          (cons j (transpose-aux i j (cdr l)))
        (if (equal (car l) j)
            (cons i (transpose-aux i j (cdr l)))
          (cons (car l) (transpose-aux i j (cdr l)))))
    ()))
(defund transpose (i j n) (transpose-aux i j (ninit n)))
```

We define a predicate that characterizes suitable arguments of `transpose`:

```
(defun trans-args-p (i j n)
  (and (posp n) (natp i) (natp j) (< i n) (< j n) (not (= i j))))
```

A transposition is a group element of ord 2:

```
(defthmd transpose-involution
  (implies (trans-args-p i j n)
           (equal (comp-perm (transpose i j n) (transpose i j n) n)
                  (ninit n))))
```

We need a predicate that recognizes a permutation as a transposition when the interchanged indices are unknown. First we define a function that identifies the least index that is not fixed by a given non-trivial permutation `p`:

```
(defun least-moved-aux (p k)
  (if (and (consp p) (equal (car p) k))
      (least-moved-aux (cdr p) (1+ k))
    k))
(defund least-moved (p) (least-moved-aux p 0))
```

The following predicate recognizes a transposition `p` in `(sym n)`:

```
(defund transp (p n)
  (let ((m (least-moved p)))
    (and (trans-args-p m (nth m p) n)
         (equal p (transpose m (nth m p) n)))))
(defthmd transp-transpose
  (implies (trans-args-p i j n)
           (transp (transpose i j n) n)))
```

A list of transpositions:

```
(defun trans-list-p (l n)
  (if (consp l)
      (and (transp (car l) n) (trans-list-p (cdr l) n))
    t))
```

We shall show that every element p of (sym n) is the product of a list (trans-list p n) of transpositions. Let m = (least-moved p), q = (transpose m (nth m p) n), and p$ = (comp-perm q p n). Note that (least-moved q) = m. Therefore, if k < m, then by nth-comp-perm,

```
(nth k p$) = (nth (nth k p) q) = (nth k q) = k,
```

while

```
(nth m p$) = (nth (nth m p) q) = m.
```

Thus, (least-moved p$) > m. This provides a measure for the following recursive definition:

```
(defun trans-list (p n)
  (declare (xargs :measure (nfix (- n (least-moved p)))))
  (let* ((m (least-moved p))
         (q (transpose m (nth m p) n))
           (p$ (comp-perm trans p n)))
    (if (and (posp n)
             (in p (sym n))
             (< m n))
        (cons trans (trans-list comp n))
      ())))
```

The desired result is easily proved using the induction scheme provided by the above definition:

```
(defthmd perm-prod-trans
  (implies (and (posp n) (in p (sym n)))
           (and (trans-list-p (trans-list p n) n)
                (equal (comp-perm-list (trans-list p n) n)
                       p))))
```

2.3 Parity

An element p of (sym n) may be represented in various ways as lists of transpositions, but we shall show that p determines whether the length of such a list is even or odd.

An *inversion* of p is a pair of indices (i . j) such that 0 ≤ i < j < n and (nth i p) > (nth j p). The *parity* of p is defined to be that of the number of its inversions. The formal definition is based on the list (pairs n) of pairs (i . j) such that 0 ≤ i < j < n. We extract from this list the list of inversions of p:

```
(defun invs-aux (p pairs)
  (if (consp pairs)
      (if (> (nth (caar pairs) p) (nth (cdar pairs) p))
          (cons (car pairs) (invs-aux p (cdr pairs)))
        (invs-aux p (cdr pairs)))
    ()))
(defund invs (p n) (invs-aux p (pairs n)))
```

We now define the parity of p:

```
(defund parity (p n) (mod (len (invs p n)) 2))
```

Accordingly, p is either *even* or *odd*:

```
(defund even-perm-p (p n) (equal (parity p n) 0))
(defund odd-perm-p (p n) (equal (parity p n) 1))
```

If p inverts i and j, then its inverse (inv-perm p n) inverts (nth j p) and (nth i p). It follows that p and (inv-perm p n) have the same number of inversions and therefore the same parity:

```
(defthmd parity-inv
  (implies (and (posp n) (in p (sym n)))
           (equal (parity (inv-perm p n) n)
                  (parity p n))))
```

The proof of the following formula for the parity of a product of permutations is more difficult and is omitted here (see the exposition in the comments in the proof script groups/symmetric.lisp):

```
(defthmd parity-comp-perm
  (implies (and (posp n) (in p (sym n)) (in r (sym n)))
           (equal (parity (comp-perm r p n) n)
                  (mod (+ (parity p n) (parity r n)) 2))))
```

It follows from parity-inv and parity-comp-perm that parity is preserved by conjugation:

```
(defthmd parity-conjugate
  (implies (and (posp n)
                (in p (sym n))
                (in a (sym n)))
           (equal (parity (conj p a (sym n)) n)
                  (parity p n))))
```

Note that a transposition of adjacent indices, (transpose i (1+ i) n), has exactly one inversion and is therefore odd, and every transposition is a conjugate of such a transposition:

```
(defthmd transpose-conjugate
  (implies (and (trans-args-p i j n) (< (1+ i) j))
           (equal (transpose i j n)
                  (comp-perm (transpose (1+ i) j n)
                             (comp-perm (transpose i (1+ i) n)
                                        (transpose (1+ i) j n)
                                        n)
                             n))))
```

We may conclude that every transposition is odd:

```
(defthmd transp-odd (implies (transp p n) (odd-perm-p p n)))
```

It follows that the parity of a product of a list of transpositions is that of the length of the list:

```
(defthmd parity-trans-list
  (implies (and (posp n) (trans-list-p l n))
           (equal (parity (comp-perm-list l n) n) (mod (len l) 2))))
```

In particular, this holds for the canonical factorization:

```
(defthmd parity-len-trans-list
  (implies (and (posp n) (in p (sym n)))
           (equal (parity p n) (mod (len (trans-list p n)) 2))))
```

2.4 Alternating Groups

The alternating group `(alt n)` is the subgroup of the symmetric group comprising the even permutations:

```
(defun even-perms-aux (l n)
  (if (consp l)
      (if (even-perm-p (car l) n)
          (cons (car l) (even-perms-aux (cdr l) n))
        (even-perms-aux (cdr l) n))
    ()))
(defund even-perms (n) (even-perms-aux (slist n) n))
(defsubgroup alt (n) (sym n)
  (posp n)
  (even-perms n))
```

It follows from `parity-conjugate` that `(alt n)` is a normal subgroup of `(sym n)`:

```
(defthmd alt-normal
  (implies (posp n) (normalp (alt n) (sym n))))
```

Assume `n` > 1 and let `s = (transpose 0 1 n)`, Then `s` is an odd permutation, and for any odd `p`, `(comp-perm s p n)` is even and therefore `p` belongs to `(lcoset s (alt n) (sym n))`. Thus, `(alt n)` has only two left cosets:

```
(defthmd subgroup-index-alt
  (implies (and (natp n) (> n 1))
           (equal (subgroup-index (alt n) (sym n)) 2)))
(defthmd order-alt
  (implies (and (natp n) (> n 1))
           (equal (order (alt n)) (/ (fact n) 2))))
```

3 Group Actions

3.1 Definition and the `defaction` Macro

Informally, an *action* of a group `g` on a dlist `d` is a mapping `a` that assigns to each element `x` of `g` and each member `s` of `d` a member `(act x s a g)` of `d`, satisfying two properties:

```
(act (e g) s a g) = s
(act x (act y s a g) a g) = (act (op x y g) s a g).
```

This may be viewed as a generalization of the operation of a group, which is an action of the group on its own element list. In fact, we use the same scheme for representing a group action as in the definition of a group: we define an action to be a matrix `a` of members of `d`, the first row of which is `d`, the *domain* of `a`:

```
(defmacro dom (a) `(car ,a))
```

The ith row of `a` defines the action of the ith element of `g` on the members of `d`:

```
(defund act (x s a g) (nth (ind s a) (nth (ind x g) a)))
```

where, according to the definition of `ind` [2, Sec. 1], `(ind s a) = (index s (dom a))`. Note that the first property above is automatic:

```
(act (e g) s a g) = (nth (ind s a) (nth 0 a)) = (nth (index s (dom a)) (dom a)) = a.
```

The defining predicate for an action calls the predicates `aclosedp` and `aassocp`, which exhaustively check the closure and associativity properties:

```
(defund actionp (a g)
  (and (groupp g)
       (dlistp (dom a))
       (consp (dom a))
       (matrixp a (order g) (len (dom a)))
       (aclosedp a g)
       (aassocp a g)))
```

It is easily verified that every group is indeed also an action:

```
(defthm actionp-groupp (implies (groupp g) (actionp g g)))
```

We have defined a macro for defining parametrized group actions, similar to `defgroup`:

`(defaction` *name args grp cond elts act*`)`,

where

- *grp* is the acting group;
- *cond* is a constraint that must be satisfied by the arguments *args*;
- *elts* is the domain;
- *act* is a term that specifies the action of a group element `x` on a member *s* of *elts*.

Conjugation is an important example. The form

```
(defaction conjugacy () g t (elts g) (conj s x g))
```

constructs the action `conjugacy` and proves three lemmas:

```
(DEFTHM ACTIONP-CONJUGACY
  (IMPLIES (GROUPP G) (ACTIONP (CONJUGACY G) G)))
(DEFTHM CONJUGACY-DOM
  (IMPLIES (GROUPP G)
           (EQUAL (DOM (CONJUGACY G)) (ELTS G))))
(DEFTHM CONJUGACY-ACT-REWRITE
  (IMPLIES (AND (GROUPP G) (IN X G) (IN S (CONJUGACY G)))
           (EQUAL (ACT X S (CONJUGACY G) G) (CONJ S X G))))
```

An action of a group g induces an action by any subgroup of g:

```
(defaction subaction (a g) h
  (and (actionp a g) (subgroupp h g))
  (dom a)
  (act x s a g))
```

As another example, if h is a subgroup of g, then we have an action of g on the left cosets of `h`, characterized by

```
(act x s (act-lcosets h g) g) = (lcoset (op x (car s) g) h g),
```

which is again constructed by `defaction`:

```
(defaction act-lcosets (h) g
  (subgroupp h g)
  (lcosets h g)
  (lcoset (op x (car s) g) h g))
```

3.2 Orbits and Stabilizers

If s is in the domain of an action a, the *orbit* of s is the ordered list of all r in (dom a) such that r = (act x s a g) for some x in g:

```
(defun orbit-aux (s a g l)
  (if (consp l)
      (let ((val (act (car l) s a g))
            (res (orbit-aux s a g (cdr l))))
        (if (member-equal val res)
            res
          (insert val res a)))
    ()))
(defund orbit (s a g) (orbit-aux s a g (elts g)))
```

We also define the list of all orbits of an action:

```
(defun orbits-aux (a g l)
  (if (consp l)
      (let ((res (orbits-aux a g (cdr l))))
        (if (member-list (car l) res)
            res
          (cons (orbit (car l) a g) res)))
    ()))
(defund orbits (a g) (orbits-aux a g (dom a)))
```

It is easily shown that every element of the domain belongs to its own orbit and that intersecting orbits are equal (i.e., distinct orbits are disjoint). It follows that appending the list of orbits yields a permutation of the domain:

```
(defthmd append-list-orbits
  (implies (actionp a g) (permp (append-list (orbits a g)) (dom a))))
```

Note that in the case of the conjugacy action, the orbit of a group element x is the conjugacy class (conjs x g) and the class equation [1, Sec. 8] is a special case of append-list-orbits.

The *stabilizer* of an element s of the domain of a is the ordered subgroup of g comprising all x such that (act x s a g) = s:

```
(defun stab-elts-aux (s a g l)
  (if (consp l)
      (if (equal (act (car l) s a g) s)
          (cons (car l) (stab-elts-aux s a g (cdr l)))
        (stab-elts-aux s a g (cdr l)))
    ()))
(defund stab-elts (s a g) (stab-elts-aux s a g (elts g)))
(defsubgroup stabilizer (s a) g
  (and (actionp a g) (member-equal s (dom a)))  ;constraints
  (stab-elts s a g))                            ;domain
```

If r is in the orbit of s, then for some x in g, (act x s a g) = r. The function actor is defined to produce such a value x. This gives rise to the following functions, which may be shown to be inverse bijections between (lcosets (stabilizer s a g) g) and (orbit s a g):

```
(defund lcosets2orbit (c s a g) (act (car c) s a g))
(defund orbit2lcosets (r s a g) (lcoset (actor r s a g) (stabilizer s a g) g))
```

Therefore, the lengths of these two lists are equal, and the following is a consequence of lagrange:

```
(defthmd stabilizer-orbit
  (implies (and (actionp a g) (in s a))
           (equal (* (order (stabilizer s a g)) (len (orbit s a g)))
                  (order g))))
```

Note that the centralizer of a group element x is its stabilizer under the conjugacy action, and the lemma len-conjs-cosets [1, Sec. 8] is a case of stabilizer-orbit.

3.3 Conjugation of Subgroups

The *conjugate* of a subgroup h of g by an element a of g is a subgroup comprising all conjugates of elements of h by a. We define this subgroup to have an ordered element list with respect to g, so that element lists of distinct conjugate subgroups cannot be permutations of one another:

```
(defun conj-sub-list-aux (l a g)
  (if (consp l)
      (insert (conj (car l) a g)
              (conj-sub-list-aux (cdr l) a g)
              g)
    ()))
(defund conj-sub-list (h a g) (conj-sub-list-aux (elts h) a g))
(defsubgroup conj-sub (h a) g
  (and (subgroupp h g) (in a g))
  (conj-sub-list h a g))
```

It is clear that a conjugate of h has the same order as h, and therefore if one conjugate is a subgroup of another, then they are equal. Since h itself need not be ordered with respect to g, h may not be a conjugate of itself, but the conjugate of h by (e g) (or by any element of h, for that matter) has the same elements as h:

```
(defthmd permp-conj-sub-e
  (implies (subgroupp h g)
           (permp (elts (conj-sub h (e g) g)) (elts h))))
```

Subgroup conjugation is an important example of a group action, the domain of which is a list of all conjugates of a given subgroup h of g:

```
(defun conjs-sub-aux (h g l)
  (if (consp l)
      (let ((c (conj-sub h (car l) g))
            (res (conjs-sub-aux h g (cdr l))))
        (if (member-equal c res)
            res
          (cons c res)))
    ()))
(defund conjs-sub (h g) (conjs-sub-aux h g (elts g)))
(defaction conj-sub-act (h) g (subgroupp h g) (conjs-sub h g) (conj-sub s x g))
```

We define the *normalizer* of a subgroup h of g to be the stabilizer of (conj-sub h (e g) g):

```
(defund normalizer (h g)
  (stabilizer (conj-sub h (e g) g)
              (conj-sub-act h g)
              g))
```

A subgroup is a normal subgroup of its normalizer:

```
(defthmd normalizer-normp
  (implies (subgroupp h g) (normalp h (normalizer h g))))
```

By `stabilizer-orbit`, the number of conjugates of `h` is the index of its normalizer, and therefore, `h` is normal in `g` iff `(mormalizer h g) = g`:

```
(defthmd index-normalizer
  (implies (subgroupp h g)
           (equal (len (conjs-sub h g))
                  (subgroup-index (normalizer h g) g))))
```

The normalizer of a conjugte of `h` is a conjugate of the normalizer of `h`:

```
(defthmd normalizer-conj-sub
  (implies (and (subgroupp m g)
                (member-equal c (conjs-sub m g)))
           (equal (normalizer c g)
                  (conj-sub (normalizer m g) (conjer-sub c m g) g))))
```

3.4 Induced Homomorphism into the Symmetric Group

An action `a` of a group `g` associates each element of `g` with a permutation of `(dom a)`. By identifying an element of `(dom a)` with its index in the list, we have an element of the symmetric group `(sym n)`, where `n = (len (dom a))`. If `x` is in `g` and `p` is the element of `(sym n)` corresponding to `x`, then for $0 \leq k < n$, the image of `k` under `p`, `(nth k p)`, is computed by the following:

```
(defund act-perm-val (x k a g)
  (index (act x (nth k (dom a)) a g)
         (dom a)))
```

Thus, the element of `(sym n)` corresponding to `x` may be computed recursively:

```
(defun act-perm-aux (x k a g)
  (if (zp k)
      ()
    (append (act-perm-aux x (1- k) a g)
            (list (act-perm-val x (1- k) a g)))))
(defund act-perm (x a g) (act-perm-aux x (order a) a g))

(defthmd act-perm-is-perm
  (implies (and (actionp a g) (in x g))
           (in (act-perm x a g) (sym (len (dom a))))))
(defthm act-perm-val-is-val
  (implies (and (actionp a g) (in x g) (member-equal k (ninit (order a))))
         (equal (nth k (act-perm x a g))
                (act-perm-val x k a g))))
```

It is clear that the identity of `g` corresponds to the identity of `(sym n)`, and that the group operation is preserved by this correspondence. Thus, we have a homomorphism from `g` into the symmetric group:

```
(defmap act-sym (a g)
  (elts g)
  (act-perm x a g))
(defthmd homomorphismp-act-sym
  (implies (actionp a g) (homomorphismp (act-sym a g) g (sym (order a)))))
```

The kernel of `(act-sym a g)` consists of the elements of `g` that act trivially on every element of `(dom a)`.

We have observed that every group `g` is an action of itself on its element list, with `(act g x s g) = (op x s g)`. The identity of `g` is the only element that acts trivially on every element (or, indeed, on any element). Therefore, every group `g` is isomorphic to a subgroup of `(sym (order g))`:

```
(defthm endomorphismp-act-sym-g
  (implies (groupp g) (endomorphismp (act-sym g g) g (sym (order g)))))
```

As another example, recall the action `act-lcosets` of a group `g` on the left cosets of a subgroup `h`, (Subsection 3.1). Clearly, the kernel of the homomorphism induced by this action is a subgroup of `h`:

```
(defthmd subgroup-kernel-act-cosets
  (implies (subgroupp h g)
           (subgroupp (kernel (act-sym (act-lcosets h g) g)
                              (sym (subgroup-index h g))
                              g)
                      h)))
```

This result has the following important consequence: If `p` is the least prime dividing the order of `g` and `h` is a subgroup of index `p`, then `h` is normal in `g`:

```
(defthmd index-least-divisor-normal
  (implies (and (subgroupp h g)
                (> (order g) 1)
                (equal (subgroup-index h g)
                       (least-prime-divisor (order g))))
           (normalp h g)))
```

The proof of this theorem also requires the observation that every homomorphism induces an endomorphism on the quotient of its kernel:

```
(defmap quotient-map (map g h)
  (lcosets (kernel map h g) g)
  (mapply map (car x)))
(defthmd endomorphismp-quotient-map
  (implies (homomorphismp map g h)
           (endomorphismp (quotient-map map g h) (quotient g (kernel map h g)) h)))
```

Tho prove `index-least-divisor-normal`, let

```
k = (kernel (act-sym (act-cosets h g) g) (sym (subgroup-index h g)) g).
```

We need only show that `k` and `h` have the same elements. (Since `h` need not be ordered with respect to `g`, the two subgroups may not be equal, but this will be sufficient to conclude that `h` is normal.) By `endomorphismp-quotient-map`, `(quotient g k)` is isomorphic to a subgroup of `(sym p)`, and therefore `(subgroup-index k g)` divides `(fact p)`, which implies `(subgroup-index k h)` divides `(fact (1- p))`. If `(subgroup-index k h)` > 1, then `(subgroup-index k g)` has a prime divisor `q`. Since `q` divides `(fact (1- p))`, `q` $<$ `p`. But since `q` divides `(order g)`, `q` $\geq$ `p` by assumption, a contradiction. Thus, `(subgroup-index k h)` = 1, which implies `(permp (elts k) (elts h))`.

4 Sylow Theorems

The Sylow theorems are a set of related results that provide information pertaining to the number of subgroups of prime power order of a finite group and the relations among them. These theorems form an important part of group theory, playing a critical role in the classification of finite groups.

Among these results is the statement that the order of a maximal p-subgroup of a finite group g is the maximal power of p that divides the order of g. As a first step, we shall prove that if h is a p-subgroup of g and p divides (subgroup-index p (normalizer h g)), then h is a proper subgroup of a larger p-subgroup of g, which may be constructed by first applying cauchy to construct a subgroup of (quotient (normalizer h g) h) of order p and then lifting it to g:

```
(defund extend-p-subgroup (h g p)
  (lift (cyclic (elt-of-ord p (quotient (normalizer h g) h))
                (quotient (normalizer h g) h))
        h
        (normalizer h g)))
(defthmd order-extend-p-subgroup
  (implies (and (subgroupp h g)
                (posp n)
                (elt-of-ord n (quotient (normalizer h g) h)))
           (let ((k (extend-p-subgroup h g n)))
             (and (subgroupp h k)
                  (subgroupp k g)
                  (equal (order k) (* n (order h)))))))
```

We recursively define a p-subgroup m = (sylow-subgroup g p) of g such that p does not divide the index of m in its normalizer:

```
(defun sylow-subgroup-aux (h g p)
  (declare (xargs :measure (nfix (- (order g) (order h)))))
  (if (and (subgroupp h g) (primep p)
           (divides p (subgroup-index h (normalizer h g))))
      (sylow-subgroup-aux (extend-p-subgroup h g p) g p)
    h))
(defund sylow-subgroup (g p) (sylow-subgroup-aux (trivial-subgroup g) g p))

(defthm index-sylow-subgroup
  (implies (and (groupp g) (primep p))
           (let ((m (sylow-subgroup g p)))
             (and (subgroupp m g)
                  (p-groupp m p)
                  (not (divides p (subgroup-index m (normalizer m g))))))))
```

We aim to show that p does not divide the index of m in g, i.e., (order m) is the maximal power of p that divides (order g). To this end, consider the action of g on the list of conjugates of m. This action has one orbit, the order of which is the index of the normalizer of m. We shall show that this index is congruent to 1 modulo p, and therefore not divisible by p.

Consider the restriction of this action to some p-subgroup h of g. Let c be a conjugate of m. By normalizer-conj-sub, the normalizer of c is a conjugate of the normalizer of m, and therefore the index of c in (normalizer c g) is not divisible by p.

Suppose x is an element of both h and (normalizer c g), but not an element of c. Since the order of x in g is a power of p, the order of the coset of x in (quotient (normalizer c g) c) is also a power of p, and p must divide (subgroup-index c (normalizer c g)), a contradiction. Thus, x is in h, then x is in (normalizer c g) iff x is in c.

By stabilizer-orbit, the length of the orbit of c under conjugation by h is 1 if h stabilizes c, and otherwise is divisible by p. But h stabilizes c iff h is a subgroup of (normalizer c g), and according to the above observation, this holds iff h is a subgroup of c:

```
(defthmd orbit-subaction-div-p
  (implies (and (subgroupp m g)
                (primep p)
                (p-groupp m p)
                (not (divides p (subgroup-index m (normalizer m g))))
                (subgroupp h g)
                (p-groupp h p)
                (in c (conj-sub-act m g)))
           (if (subgroupp h c)
               (equal (len (orbit c (subaction (conj-sub-act m g) g h) h)) 1)
             (divides p (len (orbit c (subaction (conj-sub-act m g) g h) h))))))
```

We first apply the above result to the case `h` = `m`. Since `m` is a subgroup of exactly 1 conjugate of `m`, there is exactly 1 orbit of length 1 and all others have length divisible by `p`:

```
(defthmd orbit-subaction-m-len-1
  (implies (and (subgroupp m g)
                (primep p)
                (p-groupp m p)
                (not (divides p (subgroup-index m (normalizer m g))))
                (in c (conj-sub-act m g)))
           (if (equal c (conj-sub m (e g) g))
               (equal (len (orbit c (subaction (conj-sub-act m g) g m) m)) 1)
             (divides p (len (orbit c (subaction (conj-sub-act m g) g m) m))))))
```

Appending all orbits yields the first Sylow theorem:

```
(defthmd sylow-1
  (implies (and (groupp g) (primep p))
           (let ((m (sylow-subgroup g p)))
             (equal (mod (len (conjs-sub m g)) p)
                    1))))
```

Since `(len (conjs-sub m g)) = (subgroup-index (normalizer m g) g)`, this length divides `(subgroup-index m g)`:

```
(defthmd sylow-2
  (implies (and (groupp g) (primep p))
           (let ((m (sylow-subgroup g p)))
             (divides (len (conjs-sub m g))
                      (subgroup-index m g)))))
```

Since `(len (conjs-sub m g)) = (subgroup-index (normalizer m g) g)` is not divisible by `p`, neither is `(subgroup-index m g)`:

```
(defthmd sylow-3
  (implies (and (groupp g) (primep p))
           (not (divides p (subgroup-index (sylow-subgroup g p) g)))))
```

The final Sylow theorem states that every `p`-subgroup of `g` is a subgroup of some conjugate of `m`. This is another consequence of `orbit-subaction-div-p`: If `h` were a counterexample to this claim, then according to `orbit-subaction-div-p`, the length of every orbit of `h` would be divisible by `p`, contradicting `mod-len-conjs-sub`.

The statement of the theorem requires the following finction, which searches a list `l` of subgroups of `g` for one that contains `h` as a subgroup:

```
(defun find-supergroup (h l)
  (if (consp l)
      (if (subgroupp h (car l))
          (car l)
        (find-supergroup h (cdr l)))
    ()))
(defthmd sylow-4
  (implies (and (groupp g) (primep p) (subgroupp h g) (p-groupp h p))
           (let* ((m (sylow-subgroup g p))
                  (k (find-supergroup h (conjs-sub m g))))
             (and (member-equal k (conjs-sub m g))
                  (subgroupp h k)))))
```

5 Simple Groups

A group is *simple* if has has no proper normal subgroup.

```
(defund proper-normalp (h g)
  (and (normalp h g) (> (order h) 1) (< (order h) (order g))))
```

Simple groups play an important role in the classification of finite groups. Since every group of prime order is simple, we focus on groups of composite order. One class of interest is that of the alternating groups. Note that `(alt 4)` is not simple, as may be verified by direct computation:

```
(defthmd alt-4-not-simple
  (proper-normalp (subgroup '((0 1 2 3) (1 0 3 2) (2 3 0 1) (3 2 1 0)) (sym 4))
                  (alt 4)))
```

However, `(alt n)` is simple for all $\texttt{n} \geq 5$. We shall prove this only for the case $\texttt{n} = 5$: `(alt 5)` is a simple group of order 60. In contrast to the more general theorem, our proof of this result is largely computational. We shall also prove, as an illustration of the Sylow theorems, that there are no simple groups of composite order less than 60.

5.1 Simplicity of `(alt 5)`

Let `h` be a normal subgroup of `g`. The function `conjs-list` [1, Sec. 8] constructs a list of the non-central conjugacy classes of `g`. We define `(select-conjs (conjs-list h) h)` to extract the conjugacy classes that are included in `h`:

```
(defun select-conjs (l h)
  (if (consp l)
      (if (in (caar l) h)
          (cons (car l) (select-conjs (cdr l) h))
        (select-conjs (cdr l) h))
    ()))
```

if we append the elements of that list together with the elements of `h` that belong to the center of `g`, we have a permutation of `(elts h)`. In the case of interest the center happens to be trivial. This gives us an expression for the order of `h`:

```
(defthmd len-select-conjs
  (implies (and (normalp h g) (equal (cent-elts g) (list (e g))))
           (equal (order h)
                  (1+ (len (append-list (select-conjs (conjs-list g) h)))))))
```

Thus, `(order h) = (1+ (len (append-list l)))` for some sublist `l` of `(conjs-list g)`. We need only compute this value for all such sublists of `(conjs-list (alt 5))` and observe that none of these values is a proper divisor of 60.

However, the function `conjs-list` is computationally impractical for a group of order 60. We define a more efficient and provably equivalent function, `conjs-list-fast`, based on a tail-recursive version of conjs. The lengths of the conjugacy classes of `(alt 5)` can be easily computed using this function:

```
(defun lens (l)
  (if (consp l)
      (cons (len (car l)) (lens (cdr l)))
    ()))
(defthmd lens-conjs-list-alt-5
  (equal (lens (conjs-list-fast (alt 5))) '(20 12 12 15)))
```

Clearly, no list of distinct members of this list has a sum that is a proper divisor of 60. Once we establish this simple fact (which requires some work), our theorem follows from `lagrange`:

```
(defthmd alt-5-simple (not (proper-normalp h (alt 5))))
```

5.2 Groups of Lesser Order

For every group `g` of composite `n` < 60, we shall construct a proper normal subgroup of `g`. We begin with the case of a prime power: `n = (expt p k)`, where `k` > 1. By `center-p-group` (book `cauchy`), `(order (center g))` > 1. If `(order (center g))` $<$ `(order g)`, then `(center (g))` is a proper normal subgroup. In the remaining case, `(center g) = g`, and hence `g` is abelian. Thus we need only show that `g` has a proper subgroup. But this follows from `cauchy`, which guarantees an element of order `p`. This leads to the following definition and lemma:

```
(defund normal-subgroup-prime-power (p k g)
  (declare (ignore k))
  (if (< (order (center g)) (order g))
      (center g)
    (cyclic (elt-of-ord p g) g)))
(defthm proper-normalp-prime-power
  (implies (and (groupp g)
                (equal (order g) (expt p k))
                (primep p)
                (natp k)
                (> k 1))
           (proper-normalp (normal-subgroup-prime-power p k g) g)))
```

The rest of the proof is based mainly on the Sylow theorems. We consider various cases according to the prime factorization of `n`. As a notational convenience, we shall denote `(sylow-subgroup g p)` by `hp` and `(len (conjs-sub hp g))` by `np`.

Suppose `n = (* p q)`, where `p` and `q` are primes and `p` $<$ `q`. By the Sylow theorems, `nq` divides `p` and `(mod nq p) = 1`. It follows that `np = 1`, which implies `(sylow-subgroup g q)` is normal in `g`.

```
(defund normal-subgroup-pq (p q g)
  (declare (ignore p))
  (sylow-subgroup g q))
(defthm proper-normalp-pq
  (implies (and (groupp g) (equal (order g) (* p q))
                (primep p) (primep q) (< p q))
           (proper-normalp (normal-subgroup-pq p q g) g)))
```

Next, we consider the case `n = (* p p q)`, where `p` and `q` are primes. We must show that either `np = 1` or `nq = 1`. Suppose not. Since `np` divides `q` and `(mod np p) = 1`, $q > p$. Since `nq` divides `(* p p)` and `(mod nq q) = 1`, `nq = (* p p)` and `q` divides `(1- (* p p))`. Thus, `q` divides either `(1- p)` or `(1+ p)`. Since $q > p$, `q = (1+ p)`, which implies `p = 2`, `p = 3`, and `n = 12`. Since `n3 = 4` and each 3-Sylow subgroup has 2 non-trivial elements, `g` has 8 elements of order 3. Since `n2` >1, `g` has more than 4 elements of order dividing 4, a contradiction.

```
(defund normal-subgroup-ppq (p q g)
  (if (normalp (sylow-subgroup g p) g)
      (sylow-subgroup g p)
    (sylow-subgroup g q)))
(defthm proper-normalp-ppq
  (implies (and (groupp g)
                (equal (order g) (* p p q))
                (primep p)
                (primep q)
                (not (equal p q)))
           (proper-normalp (normal-subgroup-ppq p q g) g)))
```

There are eight remaining cases, which are treated individually: 24, 30, 36, 40, 42, 48, 54, and 56. Consider the case `n = 24`. Assume `n2` >1 and let `h21` and `h22` be distinct members of `(conj-subs h2 g)`. Then `(order h21) = order h22) = 8`. Let `k = (group-intersection h21 h22 g)`. Then `(order k)` ≤ 4 and by `len-products` [2, Sec. 3],

`(len (products h21 h22 g)) = (/ (* (order h1) (order h2)) (order k))` ≤ 24,

which implies `(order k) = 4` and `(len (products h21 h22 g)) = 16`. By `index-least-divisor-normal` (Subsection 3.4), `k` is normal in both `h21` and `h22`. It follows that `(normalizer k g)` contains `(products h21 h22 g)`. Consequently, `(order (normalizer k g))` ≥ 16, which implies `(normalizer k g) = g` and `k` is normal in `g`. Thus, we have the following:

```
(defund normal-subgroup-24 (g)
  (let* ((h2 (sylow-subgroup g 2))
         (h21 (car (conjs-sub h2 g)))
         (h22 (cadr (conjs-sub h2 g)))
         (k (group-intersection h21 h22 g)))
    (if (normalp h2 g)
        h2
      k)))
(defthm proper-normalp-24
  (implies (and (groupp g) (equal (order g) 24))
           (proper-normalp (normal-subgroup-24 g) g)))
```

We omit the other seven cases, which use the same techniques as illustrated as above. We combine these results in a function that splits into cases corresponding to the composite integers less than 60:

```
(defund normal-subgroup (g)
  (case (order g)
    (4 (normal-subgroup-prime-power 2 2 g))
    (6 (normal-subgroup-pq 2 3 g))
    (8 (normal-subgroup-prime-power 2 3 g))
    (9 (normal-subgroup-prime-power 3 2 g))
    (10 (normal-subgroup-pq 2 5 g))
    (12 (normal-subgroup-ppq 2 3 g))
    ...
    (56 (normal-subgroup-56 g))
```

```
    (57 (normal-subgroup-pq 3 19 g))
    (58 (normal-subgroup-pq 2 29 g))))

(defthm no-simple-group-of-composite-order<60
  (implies (and (natp n) (> n 1) (< n 60) (not (primep n))
                (groupp g) (equal (order g) n))
           (proper-normalp (normal-subgroup g) g)))
```

6 Conclusion

Our survey of this rich topic is far from complete. We anticipate enhancements of the theory, such as the representation of a permutation as a product of disjoint cycles, which is required, for example, for a general proof of the simplicity of `(alt n)`. However, the intended scope of the project has essentially been realized. The combined content of the `groups` directory is a close approximation to that of an advanced undergraduate course that the author taught at The Cooper Union in the Spring of 1976.

The concluding section of Part I discusses the long-term objective of a formalization of algebraic number theory. The next steps in this direction are elementary linear algebra and Galois theory, the first of which is underway. We note one important difference between our approaches to groups, on one hand, and fields and vector spaces on the other. As we have observed, our interest in finite groups and the importance of proof by induction on the order of a group led us away from the characterization of a group by means of encapsulated constrained functions in favor of an explicit defining predicate. On the other hand, since we are interested in both infinite and finite fields (and the role of induction is less critical even in the latter case), we are instead pursuing the encapsulation approach in the formalization of fields as well as finite dimensional vector spaces.. A progress report may be expected at the next ACL2 workshop.

References

[1] David M. Russinoff: *A Formalization of Finite Froup Theory*. In: *ACL2 2022: 17th International Workshop on the ACL2 Theorem Prover and its Applications*, Austin, Texas, doi:10.4204/EPTCS.359.10.

[2] David M. Russinoff (2023): *A Formalization of Finite Froup Theory: Part II*. In: *ACL2 2023: 18th International Workshop on the ACL2 Theorem Prover and its Applications*, Austin, Texas.

A Case Study in Analytic Protocol Analysis in ACL2

Max von Hippel*
Northeastern University
Boston, Massachusetts
vonhippel.m@northeastern.edu

Panagiotis Manolios
Northeastern University
Boston, Massachusetts
p.manolios@northeastern.edu

Kenneth L. McMillan
University of Texas at Austin
Austin, Texas
kenmcm@cs.utexas.edu

Cristina Nita-Rotaru
Northeastern University
Boston, Massachusetts
c.nitarot@northeastern.edu

Lenore Zuck
University of Illinois Chicago
Chicago, Illinois
zuck@uic.edu

When verifying computer systems we sometimes want to study their asymptotic behaviors, i.e., how they behave in the long run. In such cases, we need *real analysis*, the area of mathematics that deals with limits and the foundations of calculus. In a prior work, we used real analysis in ACL2s to study the asymptotic behavior of the RTO computation, commonly used in congestion control algorithms across the Internet. One key component in our RTO computation analysis was proving in ACL2s that for all $\alpha \in [0,1)$, the limit as $n \to \infty$ of α^n is zero. Whereas the most obvious proof strategy involves the logarithm, whose codomain includes irrationals, by default ACL2 only supports rationals, which forced us to take a non-standard approach. In this paper, we explore different approaches to proving the above result in ACL2(r) and ACL2s, from the perspective of a relatively new user to each. We also contextualize the theorem by showing how it allowed us to prove important asymptotic properties of the RTO computation. Finally, we discuss tradeoffs between the various proof strategies and directions for future research.

1 Introduction

In contrast to purely mathematically oriented theorem provers, ACL2 is designed specifically with the verification of computer systems in mind. This focus manifests in a variety of places across the ACL2 software landscape, such as the automated proofs of termination based on context-calling graphs provided by ACL2s [2, 1, 7], the integration of QuickLisp, or the development of advanced string-solving capabilities [6]. It also manifests in what ACL2 lacks: e.g., ACL2 does not support irrationals such as e, π, or $\sqrt{2}$, meaning it cannot be used to reason about the reals in full generality. Although this limitation is often immaterial, it shows up when we want to study the asymptotic behaviors of computer systems, meaning how they behave in the long run. In such cases we require *real analysis*, the area of mathematics that deals with limits and the foundations of calculus. However, in general real analysis proofs are riddled with *real* numbers; e.g., the logarithm is often used in these proofs, and it is often the case that the logarithm of a rational number will be irrational. If all we want is to prove a real analysis result, we can use ACL2(r) [3], the variant of ACL2 that supports real numbers. Unfortunately, ACL2(r) proofs cannot be imported into a normal ACL2 environment because the two systems have conflicting underlying theories, e.g., in ACL2 it is a theorem that $\forall x :: x^2 \neq 2$, while in ACL2(r), $\sqrt{2}$ is a number. So, what if we have formalized and studied a computer system in ACL2, and now we want to look at its asymptotic behaviors, without needing to port our model to ACL2(r)?

*Authors are listed alphabetically by last name.

A. Coglio, S. Swords (Eds.): 18th ACL2 Workshop (ACL2-2023)
EPTCS 393, 2023, pp. 50–66, doi:10.4204/EPTCS.393.6

In this work we focus on one such scenario, drawn from our recent work [4] studying Karn's algorithm [5] and the related Retransmission TimeOut (RTO) computation [9]. This work can be viewed as a companion paper to our work in [4], with a special focus on the proof strategy for the RTO observations. Next, we briefly summarize our work in [4] as it provides context for this study.

1.1 Motivating Example

In order to understand the RTO computation and its motivation, we first need to understand the context in which it is used. In the real world this context is an Internet connection between two computers, such as might occur between a sender and receiver using TCP. In the most general sense, the context consists of two endpoints communicating over a *channel*. The endpoints send and receive *datagrams* partitioned into *packets* (that the sender transmits to the receiver) and *acknowledgments*, or ACKs (that the receiver transmits to the sender), and each datagram is uniquely identifiable by its natural id and type (packet or ACK). The channel is responsible for delivering messages transmitted from one endpoint to the other. However, it might not do so reliably. It cannot create new messages, but may reorder, drop, delay, or duplicate transmitted ones.[1]

We assume the sender does not transmit a packet $p > 1$ unless it previously transmitted all packets in $[1, p-1]$, although it may transmit $p = 1$ at any time. We additionally assume that ACKs are cumulative in the sense that, the receiver does not transmit an ACK $a > 1$ unless it was previously delivered packets $1, 2, \ldots, a-1$ but not a. In the event that the receiver cannot cumulatively acknowledge anything, for instance, if it was delivered the packet 2 but never 1, then it may transmit the trivial acknowledgment of 1 (which does not acknowledge any packets). We also assume the receiver transmits an ACK whenever it is delivered a packet. When the sender receives an acknowledgment a, it considers a to be *new* iff a exceeds all the ACKs it received previously. In other words, a is new iff it acknowledges at least one previously un-acknowledged packet.

In the real world, the channel may have limited bandwidth, and will start to lose datagrams when its queues become full[2]. This bandwidth is unknowable to either endpoint, so the sender is forced to use ACKs to assess the instantaneous state of the channel and react accordingly. It does so in two ways. First, the sender measures the round-trip time (RTT) using its local clock between when it first transmits a packet p and when it first receives any acknowledgment $a > p$, as an indication of the pace at which the channel is delivering datagrams. It can use this information to moderate its transmission rate. Second, if no new ACKs arrive for some amount of time, the sender can assume the channel has been overwhelmed and is dropping data. In this case, it can slow its pace of transmission, and begin retransmitting unacknowledged data accordingly. The time the sender will wait before timing out, slowing its pace, and retransmitting, is called the RTO and defined in RFC6298 [9].

The conjunction of these two mechanisms creates a problem: suppose the sender retransmits an unacknowledged packet p, then receives some new ACK $a > p$. How does it know which transmission of p triggered the ACK? The estimated RTT will differ depending on the answer. We illustrate this situation in Figure 1[3]. One solution, known as Karn's algorithm, is to only sample RTTs for packets that were transmitted precisely once, since ACKs for these packets are unambiguous [5].

We formally modeled Karn's algorithm in Ivy [8] with the network model outlined above. We proved various inductive invariants about the algorithm, including that it samples a real RTT, that this RTT is in some sense pessimistic, and that when the receiver-to-sender channel path is FIFO, this RTT is for the

[1] This model is less strict than the typical IP one where duplication is disallowed.

[2] Such lossy communication is commonly modeled using a so-called "token bucket filter".

[3] Icons are from https://openmoji.org/

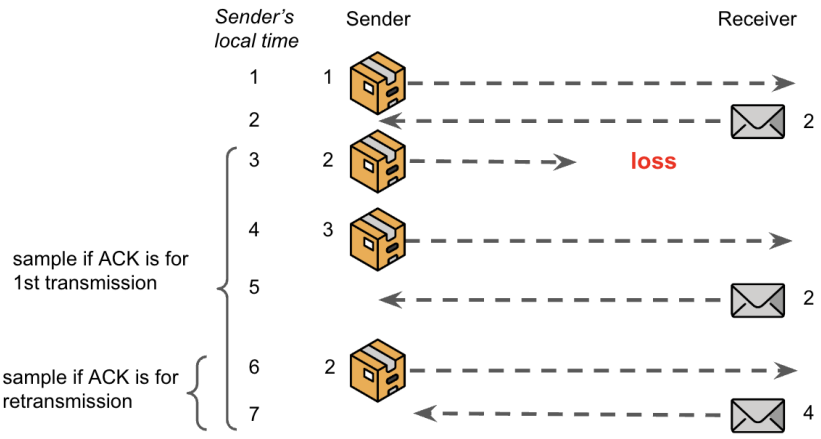


Figure 1: Message sequence chart illustrating an ambiguous ACK, with the sender's local clock shown on the left. Sender's packets are illustrated as packets, while receiver's ACKs are shown as envelopes. The first time the sender transmits 2 the packet is lost in-transit. Later, upon receiving a cumulative ACK of 2, the sender determines the receiver had not yet received the 2 packet and thus the packet might be lost in transit. It thus retransmits 2. Ultimately the receiver receives the retransmission and responds with a cumulative ACK of 4. When the sender receives this ACK it cannot determine which 2 packet delivery triggered the ACK transmission and thus, it does not know whether to measure an RTT of 7-3=4 or 7-6=1. Hence, the ACK is ambiguous, so any sampled RTT would be as well.

packet whose id is equal to the previously highest-received acknowledgment. Then, we formally modeled the RTO computation in ACL2s. We chose ACL2s over Ivy because the computation is defined over real numbers, which we chose to model as rationals, and Ivy only supports integers.[4] And in particular, we chose ACL2s over ACL2 because we made frequent use of its features. For example, we use the built-in counterexample generation to show that a variable referred to as the "RTT variance" is not actually a statistical variance; and in one of our proofs, we use an automated proof of function termination to prove the existence of a particular value (namely, the value returned by the function in question).

The RTO computation is recursively defined over the RTT samples $\mathsf{S}_1, \mathsf{S}_2, \ldots$ output by Karn's algorithm and parameterized by three positive constants ($\alpha < 1$, $\beta < 1$, and G) as follows.

$$\begin{aligned} \mathsf{rto}_i &= \mathsf{srtt}_i + \max(G, 4 \cdot \mathsf{rttvar}_i) \\ \mathsf{rttvar}_i &= \begin{cases} \mathsf{S}_i/2 & \text{if } i = 1 \\ (1-\beta)\mathsf{rttvar}_{i-1} + \beta|\mathsf{srtt}_{i-1} - \mathsf{S}_i| & \text{if } i > 1 \end{cases} \\ \mathsf{srtt}_i &= \begin{cases} \mathsf{S}_i & \text{if } i = 1 \\ (1-\alpha)\mathsf{srtt}_{i-1} + \alpha\mathsf{S}_i & \text{if } i > 1 \end{cases} \end{aligned} \tag{1}$$

Note, we use "RTO" when discussing the calculation generally, and "rto" when discussing its actual implementation (given in the equation above).

We looked at what we called the *steady-state* where for some rational *center* c and *radius* r, the samples $\mathsf{S}_i, \mathsf{S}_{i+1}, \ldots, \mathsf{S}_{i+n}$ all fall within the bounds $[c-r, c+r]$, and we proved the following.

[4] In practice, the implementations we are aware of use integers, however, when Karn and Partridge wrote the algorithm down on paper, they did so using reals.

I. The srtt_{i+n} is bounded by the interval $[L,H]$ defined as follows.

$$\begin{aligned} L &= (1-\alpha)^{n+1}\mathsf{srtt}_{i-1} + (1-(1-\alpha)^{n+1})(c-r) \\ H &= (1-\alpha)^{n+1}\mathsf{srtt}_{i-1} + (1-(1-\alpha)^{n+1})(c+r) \end{aligned} \tag{2}$$

II. For all $0 \leq m < n$, rttvar_{i+n} is upper-bounded by the following expression.

$$\begin{aligned} &(1-\beta)^{n+1-m}\mathsf{rttvar}_{i+m-1} + (1-(1-\beta)^{n+1-m})\Delta_m \text{ , where} \\ &\Delta_m = (1-\alpha)^{m+1}\mathsf{srtt}_{i-1} + 2r - (1-\alpha)^{m+1}(c+r) \end{aligned} \tag{3}$$

Then we reanalyzed the results in the asymptotic case. In other words, we asked what these bounds converge to as the number n of consecutively bounded samples, as well as the cutoff $m < n$ for the bound Δ_m defined above, grow toward infinity. By *limit*, we are referring to the standard definition[5] from real analysis, which we give below using the 1D Euclidean metric $d(x,y) = |x-y|$.

Definition 1 (Limit to ∞)**.** *Let* $(a_i)_{i=0}^{\infty} \in \mathbb{R}^{\omega}$ *and* $\ell \in \mathbb{R}$*. Then* $\lim_{i\to\infty} a_i = \ell$ iff *the following holds:*

$$\forall \varepsilon > 0 :: \exists \delta > 0 :: \forall n > \delta :: |a_n - \ell| < \varepsilon$$

We first proved the following theorem, which is the focus of this paper.

Theorem 1. $\forall \alpha \in [0,1) :: \lim_{n\to\infty} \alpha^n = 0$

Theorem 1 can be manually proven as follows.

Proof. Let $\varepsilon > 0$ and $0 \leq \alpha < 1$ arbitrarily. If $\alpha = 0$ the result is immediate; suppose $\alpha > 0$. Suppose $\varepsilon < 1$, noting that if the theorem holds for $\varepsilon < 1$ then it holds for $\varepsilon \geq 1$. Let $\delta = \ln(\varepsilon)/\ln(\alpha)$. Note that $\ln(\varepsilon)$ and $\ln(\alpha)$ are negative. Let n be some natural number and observe that $n\ln(\alpha) = \ln(\alpha^n)$. Thus:

$$\begin{aligned} n > \delta &\iff n > \ln(\varepsilon)/\ln(\alpha) && \text{by definition of } \delta \\ &\iff n\ln(\alpha) < \ln(\varepsilon) && \text{multiplying each side by } \ln(a) \\ &\iff e^{n\ln(\alpha)} < e^{\ln(\varepsilon)} && \text{raising each side above } e \\ &\iff e^{\ln(\alpha^n)} < \varepsilon && \text{because } e^{\ln(x)} = x \text{ for all } x, \text{ and } n\ln(\alpha) = \ln(\alpha^n) \\ &\iff \alpha^n < \varepsilon && \text{because } e^{\ln(x)} = x \text{ for all } x \end{aligned}$$

□

We then used this theorem to show that $\lim_{n\to\infty} L = c - r$, $\lim_{n\to\infty} H = c + r$, and the limit as n and $m < n$ both grow toward infinity of the upper bound in Eqn. 3 is precisely $2r$.

[5] Limit proofs that assume an $\varepsilon > 0$ and then prove the existence of a corresponding $\delta > 0$ satisfying this definition are commonly reffered to as ε/δ-proofs.

Outline. The rest of this paper is organized as follows. We study Theorem 1 in Sections 2 and 3. Specifically, in Section 2 we formalize the English-language proof given above in ACL2(r). But recall, we cannot use an ACL2(r) proof to study the RTO system we defined in ACl2s, without totally remodeling it, as the two proof systems are incompatible. Thus in order to have our asymptotic proofs in the same model as our other preexisting proofs about the RTO system, we need a rational proof. We give two such proofs in Section 3. The first uses the ceiling function to define a δ directly. The second begins by proving that $\lim_{n\to\infty} 1/2^n = 0$, then uses the binomial theorem to construct a δ such that $n > \delta \implies \alpha^\delta < 1/2$. For the second proof strategy, we show two different ways to prove $\lim_{n\to\infty} 1/2^n = 0$, one of which is more automatic than the other. With those proofs out of the way, we show how to derive the limits for the bounds on srtt and rttvar in Section 4, which concludes our analytic study of the RTO. We discuss trade-offs between the various proofs and lessons learned in Section 5 and conclude in Section 6.

2 Real Proof

In this Section we overview the most obvious proof strategy for Theorem 1 – the one we gave in the introduction – and its formalization in ACL2(r), the variant of ACL2 that supports real numbers. Because the rationals form a dense subset of the reals, this proof implies the desired result over the rationals as well. However, since ACl2 and ACL2(r) are theoretically incompatible, we cannot just import the proof into our preexisting ACL2s model to cohabitate with our other theorems about the RTO calculation.

The theorem we aim to prove uses an existential quantifier, so we define it via `defun-sk`. Note that our theorem statement will be the same in Section 3, except that since we will be using ACL2s in those proofs, we will also have type declarations and guards there. Notice how we can drop the absolute value signs from Definition 1 because α is assumed to be positive, implying that α^n is also positive.

```
(defun-sk lim-0 (a e n)
  (exists (d)
    (=> (^ (realp e) (< 0 e) (< d n)) (< (raise a n) e))))

(defthm lim-a^n->0
  (=> (^ (realp a) (< 0 a) (< a 1) (realp e) (< 0 e) (natp n))
      (lim-0 a e n)) :instructions ...) ;; proof will go here
```

The most important step in an ε/δ proof is defining the δ. We do so by defining a witness function $d_a : \varepsilon \to \delta$, so that the proof obligation reduces to showing $\forall \varepsilon > 0 :: \forall n > d_a(\varepsilon) :: \alpha^n < \varepsilon$.

```
(defun d (eps a) (/ (acl2-ln eps) (acl2-ln a)))
```

The remainder of the proof consists of two important steps. First, we define a number of arithmetic lemmas which ACL2s proves automatically. Second, because ACL2(r) lacks a generic real logarithm or exponent (having only the natural variants), we prove a translational lemma (R1) saying that $e^{n\ln(\alpha)} = \alpha^n$. Then we rephrase the proof from §1 in terms of e, at which point it goes through easily.

2.1 Arithmetic Lemmas

We began by proving some basic arithmetic lemmas, which we needed for the more complicated proofs. We proved that if $e^y < 1$ then $y < 0$, and that if $y \in (0,1)$, then $\ln(y) < 0$. Then we proved two facts about fractions of logarithms. First, if $\alpha \in (0,1)$ then $(\ln(\varepsilon)/\ln(\alpha))\ln(\alpha) = \ln(\varepsilon)$. Second, if ε and α both fall within $(0,1)$ and $n > \ln(\varepsilon)/\ln(\alpha)$, then $n\ln(\alpha) < \ln(\varepsilon)$, and thus, since the natural exponent is

monotonic (i.e., $x < y \implies e^x < e^y$), it follows that $e^{n\ln(\alpha)} < e^{\ln(\varepsilon)} = \varepsilon$. Finally, combining these results gave us that if α and $n \in \mathbb{R}_+$ then $\ln(\alpha^n) = \ln(e^{n\ln(\alpha)}) = n\ln(\alpha)$.

2.2 Translational Lemma

Our arithmetic lemmas allow us to prove the following translational result.

Lemma R1. *For all positive reals α and n, $e^{n\ln(\alpha)} = \alpha^n$.*

2.3 Proof of Theorem 1

Next we prove the desired result without loss of generality, and without explicitly using quantifiers. Our proof uses Lemma R1 as a hint.

Lemma R2. *Let α and ε be reals in $(0,1)$ and let $n > d_\alpha(\varepsilon)$ be a natural. Then $\alpha^n < \varepsilon$.*

Notice how this immediately gives us our desired result, because if $\varepsilon < 1$ then we can just use Lemma R2 directly, and if $\varepsilon \geq 1$, we can use it with a new "epsilon" $< \varepsilon$. And this is precisely the strategy we take in the `instructions` to our proof of Theorem 1, which go as follows:

i. Promote all variables then case-split on $\varepsilon < 1$.

ii. Case 1: $\varepsilon < 1$. Use Lemma R2. Then instantiate `lim-0-suff` with $\delta = d_\alpha(\varepsilon)$ and prove.

iii. Case 2: $\varepsilon > 1$. Use Lemma R2 with a new "epsilon" value of $\varepsilon' = 1/2$ and claim that all its preconditions are satisfied. Further claim that if $d_\alpha(1/2) < n$ then $\alpha^n < 1/2 < \varepsilon$. Then instantiate `lim-0-suff` with $\delta = d_\alpha(1/2)$, promote, and prove.

On the one hand, our proof is straightforward in the sense that it mostly follows the English-language proof we outlined in the introduction. On the other hand, we can easily see some places where more machinery and theorems in the nonstandard arithmetic library would drastically simplify things. The biggest ommision is that the nonstandard arithmetic library only supports natural exponent and logarithm, which forced us to use the translational lemma. Most interestingly, this is not the shortest proof in this paper! There is actually a more concise[6], rational proof which we outline in the next Section.

3 Rational Proofs

In this Section, we reprove Theorem 1 in ACL2s, using only rationals. We present two proofs. The first is the one we used in our motivating work [4], where we explicitly construct the δ using the ceiling function. To do so, we have to prove various properties of that function. The second proof is our most concise, and proceeds in two steps. First we show that $\lim_{n\to\infty} 1/2^n = 0$. Then using the binomial theorem, we show how, for any $0 < \alpha < 1$, we can construct an n such that $\alpha^n < 1/2$. The result follows. For convenience, we refer to the first proof (given in §3.1) as the *ceiling proof* and the second (§3.2) as the *binomial proof*. We recap in §3.3.

3.1 Ceiling Proof

In prose, the ceiling proof of Theorem 1 proceeds as follows.

[6](as measured by lines of code and number of imported books)

Proof. Let $0 \leq \alpha < 1$ and $\varepsilon > 0$, arbitrarily. Let $k = \lceil a/(1-a) \rceil$ and observe that $a \leq k/(k+1)$. Let $f(n) = k\alpha^k/n$. As an intermediary lemma, we claim that for all $n \geq k$, $\alpha^n \leq f(n)$.

Base Case: $n = k$ thus $f(n) = \alpha^k \geq \alpha^n$ and we are done.

Inductive Step: By inductive hypothesis, we have

$$a^n \leq k\alpha^k/n \tag{4}$$

and $k \leq n$. This gives us $k/(k+1) \leq n/(n+1)$ and thus:

$$\alpha \leq n/(n+1) \tag{5}$$

Multiplying Eqn. 4 through by α, we get $\alpha^{n+1} \leq k\alpha^{k+1}/n$. Combining this with Eqn. 5:

$$\alpha^{n+1} \leq (k\alpha^k/n)\frac{n}{n+1} = k\alpha^k/(n+1) \tag{6}$$

and we are done.

Hence induction: $\forall n \geq k$, $\alpha^n \leq f(n)$. Now, let $\delta = \lceil k\alpha^k/\varepsilon \rceil$. It follows that $\forall n \geq \delta$, $f(n) \leq \varepsilon$, and thus by the above result, $\alpha^n \leq \varepsilon$. We get $\alpha^n < \varepsilon$ by repeating this process for $\varepsilon/2$, and we are done. □

Although the proof is relatively straightforward on paper, as we will see, it is much more challenging in ACL2s. The primary issue is that ACL2/ACL2s does not by default know very much about the ceiling function, so we will be forced to prove many obvious lemmas before making the essential argument. Since we are in ACL2s now, we first restate Theorem 1 with types.

```
(defun-sk lim-0 (a e n)
  (declare (xargs :guard (and (posratp a) (< a 1) (posratp e) (natp n))
                  :verify-guards t))
  (exists (d) (and (natp d) (implies (< d n) (< (expt a n) e)))))

(property lim-a^n->0 (a e :pos-rational n :nat)
  :hyps (< a 1)
  (lim-0 a e n) :instructions ...) ;; proof will go here
```

The rest of the subsection is organized in follows. We prove arithmetic lemmas in §3.1.1. We use these lemmas to prove the "intermediary lemma" in §3.1.2, which we use to prove prove Thm. 1 in §3.1.3.

3.1.1 Arithmetic Lemmas

In order to fill out the `instructions`, we first need some lemmas, primarily about the ceiling function.

Lemma C1. *For all $x, y \in \mathbb{Q}_+$, if $\lceil x \rceil < \lceil y \rceil$ then (i) $x \leq \lceil x \rceil$ and (ii) $\lceil x \rceil < y$.*

For the next Lemma we write a manual proof, adapted from [10], and utilizing Lemma C1, which we give immediately below.

Lemma C2. *Let $m, n \in \mathbb{N}_+$ and $x \in \mathbb{Q}_+$. Then $\lceil x/mn \rceil = \lceil \lceil x/m \rceil / n \rceil$.*

Proof. Observe:

$$\lceil x/m \rceil - 1 < x/m \leq \lceil x/m \rceil \tag{7}$$

Dividing Eqn. 7 by n, we get $(\lceil x/m \rceil - 1)/n < x/mn \leq \lceil x/m \rceil/n$. We also observe that $\lceil x/mn \rceil \leq \lceil \lceil x/m \rceil/n \rceil$. Thus by Lemma C1:

$$x/mn \leq \lceil x/mn \rceil < \lceil x/m \rceil/n \tag{8}$$

Suppose (for a contradiction) that $\lceil x/mn \rceil < \lceil \lceil x/m \rceil/n \rceil$. Multiplying Eqn. 8 by n, we get $x/m \leq n\lceil x/mn \rceil < \lceil x/m \rceil$, which is sufficient information for ACL2s to automatically find the contradiction:

$$\lceil x/m \rceil \leq n\lceil x/mn \rceil < \lceil x/m \rceil \tag{9}$$

sufficing to show that $\lceil x/mn \rceil \not< \lceil \lceil x/m \rceil/n \rceil$. But then $\lceil x/mn \rceil = \lceil \lceil x/m \rceil/n \rceil$ and we are done. □

Next we observe that for all $x, y, z \in \mathbb{Q}_+$, if $y \leq z$ then $y/x \leq z/x$ and moreover, $yx \leq zx$. The following property of the ceiling function automatically follows. Next we make an important observation about the ceiling function.

Lemma C3. *Let $\alpha \in \mathbb{Q}$ such that $0 < \alpha < 1$. Let $k = \lceil \alpha/(1-\alpha) \rceil$. Then $\alpha \leq k/(1+k)$.*

Proof. First note that $\alpha/(1-\alpha) \leq k$. Observe that for all $x, y, z \in \mathbb{Q}_+$, if $y \leq z$, then $yx \leq zx$. It follows that $\alpha = (\alpha/(1-\alpha))(1-\alpha) \leq k(1-\alpha)$. Next observe that $\alpha \leq k(1-\alpha) = k - k\alpha$. Adding $k\alpha$ to each side, we get $\alpha + k\alpha = \alpha(1+k) \leq k$. Again consider $x, y, z \in \mathbb{Q}_+$ such that $y \leq z$, but this time, observe that $yx \leq zx$. Thus, $\alpha(1+k)/(1+k) = \alpha \leq k/(1+k)$, and we are done. □

Next we prove two lemmas about fractions.

Lemma C4. *For all $k \in \mathbb{N}_+$ and $\alpha \in \mathbb{Q}_+$, $k\alpha^k/k = \alpha^k$.*

Lemma C5. *For all $k \leq n \in \mathbb{N}, k/(1+k) \leq n/(1+n)$.*

Finally, we make some obvious arithmetic observations, leading to the following result.

Lemma C6. *For all $x, y \in \mathbb{Q}_+$, we have $x/\lceil x/y \rceil \leq y$.*

Proof. Note $x/y \leq \lceil x/y \rceil$, thus $1/\lceil x/y \rceil \leq 1/(x/y)$. Multiply both sides by x, and we are done. □

3.1.2 Inductive Proof of Intermediary Lemma

With these arithmetic lemmas completed we can move on to the actual proof. For convenience, we will define a parameterized function $f_\alpha : \mathbb{N}_+ \to \mathbb{Q}_+$ such that $f_\alpha(n) = k\alpha^k/n$ for $k = \lceil \alpha/(1-\alpha) \rceil$. As an intermediary lemma, we claim that for all $n \geq k$, $a^n \leq f(n)$. Assuming the lemma holds, we can let $\delta = \lceil ka^k/\varepsilon \rceil$, and we immediately get that for all $n \geq \delta$, $a^n \leq f_\alpha(n) \leq \varepsilon$. Theorem 1 immediately follows. This sub-subsection is spent proving the intermediary lemma.

Lemma C7 (Base Case)**.** *Let $\alpha \in \mathbb{Q}_+$ and let $k = \lceil \alpha/(1-\alpha) \rceil$. Then $f_\alpha(k) = a^k$.*

Proof. Follows directly from Lemma C4. □

Before the inductive step, we need one more helper lemma, the proof of which follows from our prior arithmetic observations.

Lemma C8. *For all $n, k \in \mathbb{N}_+$ and $\alpha \in \mathbb{Q}_+$, if $\alpha^{n+1} \leq \alpha k\alpha^k/n$, then $\alpha^{n+1} \leq ka^k/(1+n)$.*

Lemma C9 (Inductive Step)**.** *Let $\alpha \in \mathbb{Q}_+$ and $n \in \mathbb{N}$. Suppose that $\alpha < 1$, $k = \lceil \alpha/(1-\alpha) \rceil \leq n$, and $\alpha^n \leq f_\alpha(n)$. Then $\alpha^{1+n} \leq f_\alpha(1+n)$.*

Proof. By Lemma C7, $f_\alpha(k) = a^k$. By Lemma C5, $\alpha \leq k/(1+k) \leq n/(1+n)$. Then by our arithmetic observations, $\alpha^{n+1} \leq f_\alpha(k)\alpha$, and thus, $\alpha^{n+1} \leq ka^k/(1+n) = f_\alpha(1+n)$. □

Although at this point we've laid out our inductive argument, we still need to implement it in ACL2s. To begin with, this means defining an inductive scheme.

```
(definec ikn (a :pos-rational n :nat) :nat
  :ic (< a 1)
  (if (> (ceiling (/ a (- 1 a)) 1) n) 0 (1+ (ikn a (- n 1)))))
```

We use this scheme in the proof of the next lemma.

Lemma C10 (Intermediary Lemma)**.** *Let $\alpha < 1$ be in $\mathbb{Q}_+$ and $n \geq \lceil \alpha/(1-\alpha) \rceil$ in $\mathbb{N}$. Then $\alpha^n \leq f_\alpha(n)$.*

Proof. Induct on `ikn`. Use Lemma C7 for the base case and Lemma C9 for the inductive step. □

3.1.3 Proof of Theorem 1

Our proof strategy is as follows. First, we introduce a function $\delta_\alpha : \mathbb{R}_+ \to \mathbb{N}_+$ defined by $\varepsilon \mapsto \max\{k,d\}$, where $k = \lceil \alpha/(1-\alpha) \rceil$ as before, and $d = \lceil k\alpha^k/\varepsilon \rceil$. Then we prove three lemmas about this function (given immediately below) which together suffice to imply Theorem 1. We use k and d as defined above.

Lemma C11. *Let $\alpha < 1$ and ε be in $\mathbb{Q}_+$ and $n \in \mathbb{N}$. Suppose $\delta_\alpha(\varepsilon) \leq n$. Then $k \leq n$.*

Proof. By definition of δ_α, we have $\max\{k,d\} \leq n$. Since $k \leq \max\{k,d\}$, we are done. □

Lemma C12. *Let $\alpha < 1$ and ε be in $\mathbb{Q}_+$ and $n \in \mathbb{N}$. Suppose $\delta_\alpha(\varepsilon) \leq n$. Then $\alpha^n \leq f_\alpha(n)$.*

Proof. Follows automatically from Lemmas C10 and C11 with the definitions of f_α and δ_α. □

Lemma C13. *Let $\alpha < 1$ and ε be in $\mathbb{Q}_+$ and $n \in \mathbb{N}$. Suppose $\delta_\alpha(\varepsilon) \leq n$. Then $f_\alpha(n) \leq \varepsilon$.*

Proof. By the definition of δ_α, we have $\max\{k,d\} \leq n$. Our prior arithmetic observations give us that $k\alpha^k/n \leq k\alpha^k/d$. Since $d = \lceil k\alpha^k/\varepsilon \rceil$, by Lemma C6, clearly $k\alpha^k/d \leq \varepsilon$. The result follows. □

Armed with these Lemmas, we can prove a "helper lemma" like we did previously. But in this case, we use $\leq$ instead of $<$ because of the way we structured our argument based on the intermediary result.

Lemma C14. *For all $\alpha < 1$ and ε in $\mathbb{Q}_+$ and $n \in \mathbb{N}_+$, if $\delta_\alpha(\varepsilon) \leq n$ then $\alpha^n \leq \varepsilon$.*

Proof. Follows from Lemmas C12 and C13 after observing that all their preconditions are met. □

Finally, we can provide the `instructions` to prove Theorem 1. Note how we divide ε by 2 in order to transform $\leq$ into $<$, to fit Definition 1.[7]

```
((:use (:instance lim-0-suff (d (delta a (/ e 2)))))
 (:use (:instance a^n->0 (a a) (e (/ e 2)) (n n)))
 :pro :prove)
```

[7] N.b., this trick suffices to show that the alternative definition with $\leq$ is equivalent.

3.2 Binomial Proof

In this subsection, we propose an alternative proof. The strategy can be split into two steps. First, we prove that $0 \leq \alpha \leq 1/2 \implies \lim_{n\to\infty} \alpha^n = 0$. Second, we prove that for all $\alpha \in [0,1)$, there exists some $\delta \in \mathbb{N}$ such that $n > \delta \implies \alpha^n \leq 1/2$. We rely on the binomial theorem to find this δ, hence the name of the proof. These results suffice to prove Theorem 1.

We found two ways to approach the first step. The first way was to attack the problem directly, with an ε/δ proof. The second was to leverage the termination analysis in ACL2s to find a δ semi-automatically. We cover the first approach in §3.2.1 and the second in §3.2.2. Then in §3.2.3 we show how, given either approach, we can prove Thm. 1 by completing the "second step" described above.

3.2.1 Manual Proof of $0 \leq \alpha \leq 1/2 \implies \lim_{n\to\infty} \alpha^n = 0$.

We begin by importing the proof-by-arithmetic book.

```
(include-book "make-event/proof-by-arith" :dir :system)
```

We then prove the a sequence of simple arithmetic facts, using some combination of the `linear`, `match-free`, and `all` rule classes. First, like we did in the prior section, we show that the exponent is monotonic. Second, we show that for all $n \in \mathbb{N}$, $n < 2^n$, and thus, if n is positive, then $1/2^n < 1/n$. Then we introduce an arithmetic trick by which we can extract a number smaller than ε, namely, if $\varepsilon = x/y$ is a positive rational, then $1/y < \varepsilon$. Combining these results yields the following two lemmas.

Lemma BM1. *For all $\alpha \leq 1/2$ in $\mathbb{Q}_+$, and for all $d \in \mathbb{N}_+$, $\alpha^d \leq 1/2^d$.*

Lemma BM2. *For all $\alpha, \varepsilon = x/y \in \mathbb{Q}_+$, where $x, y \in \mathbb{N}_+$, if $\alpha \leq 1/2$, then $\alpha^y \leq \varepsilon$.*

At this point, the desired result follows directly from Lemma BM2.

3.2.2 Semi-Automatic Proof of $0 \leq \alpha \leq 1/2 \implies \lim_{n\to\infty} \alpha^n = 0$.

In the semi-automatic proof, we begin by defining two functions. The first function, $\mu : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, is defined by $(b,q) \mapsto b$ *if* $q < 2^b$ *else* $\mu(b+1,q)$, and can be viewed as a "helper function" to the second, $d : \mathbb{Q}_+ \to \mathbb{N}$, which is defined by $\varepsilon \mapsto \mu(0, \text{denominator}(\varepsilon))$. When we define these two functions, ACL2s automatically proves that they terminate, meaning that for all possible inputs of $\varepsilon \in \mathbb{Q}_+$, $\mu(0,\varepsilon)$ terminates. We use this with an inverse argument to show $1/2^{d(\varepsilon)} \leq \varepsilon$, which we then manipulate to get the desired result. Because the argument and manipulation require writing down additional lemmas, we call this proof *semi*-automatic. Next, we prove three lemmas about these functions. The first two go through automatically, whereas the third requires a manual proof.

Lemma BA1. *For all $b, q \in \mathbb{N}$, $q < 2^{\mu(b,q)}$.*

Lemma BA2. *For all $q \in \mathbb{N}_+$, $1/2^{\mu(0,q)} < 1/q$.*

Lemma BA3. *For all $\varepsilon > 0$, $1/2^{d(\varepsilon)} < \varepsilon$.*

Proof. First observe that $1/\text{denominator}(\varepsilon) \leq \varepsilon$. Then observe that $1/2^{\text{denominator}(\varepsilon)} < 1/\text{denominator}(\varepsilon)$. The rest follows automatically. □

Next, we establish the monotonicity of the exponent, both strictly ($<$) and otherwise ($\leq$). This allows us to prove the following.

Lemma BA4. *For all $k \leq n \in \mathbb{N}_+$ and $\alpha < 1$ in $\mathbb{Q}_+$, $\alpha^n \leq \alpha^k$.*

Lemma BA5. *For all $\varepsilon > 0$ in $\mathbb{Q}_+$ and $n > d(\varepsilon)$ in $\mathbb{N}$, $1/2^n < \varepsilon$.*

Proof. Follows from the (non-strict) monotonicity of the exponent, Lemma BA3, and the observation that for all $n \in \mathbb{N}$, $1/2^n = (1/2)^n$. □

We prove the next lemma in its given form, and rewritten using *numerator* and *denominator*.

Lemma BA6. *For all $p,q \in \mathbb{N}_+$, if $p/q < 1$ then $p < q$.*

Lemma BA7. *For all $x \leq y$ and α in $\mathbb{N}_+$, $\alpha/y \leq \alpha/x$.*

Finally, we get the desired result.

Lemma BA8. *For all $\alpha \leq 1/2$ and ε in $\mathbb{Q}_+$, and for all $n \geq d(\varepsilon)$ in $\mathbb{N}$, $\alpha^n < \varepsilon$.*

Proof. Follows from Lemmas BA5 and BA7. □

Having shown two ways to derive the first step of our outlined proof strategy, we now move on to the second step, where we invoke the binomial theorem.

3.2.3 Remainder of Binomial Proof

The remainder of the proof begins with the following lemma. The lemma is the same regardless of which strategy we take for part 1 (i.e., manual, or semi-automatic), however, its proof differs slightly with each choice. Thus for brevity, we only include the proof assuming we took the semi-automatic approach, since it is the most recent in the text and thus the easiest to compare to here. The alternative version given the manual approach is very nearly identical.

Lemma BB1. *For all $\alpha = x/y \in \mathbb{Q}_+$, where $x,y \in \mathbb{N}_+$, $\alpha \leq x/(1+x)$.*

Proof. Follows from Lemmas BA6 and BA7. □

Next we import the binomial theorem into our proof.

```
(include-book "arithmetic/binomial" :dir :system)
```

Note that the `binomial` book was written in ACL2. Since we are in ACL2s, we have an additional obligation to check types. So, we prove four convenient lemmas about the types involved in the binomial expansion as defined in that book.

Lemma BB2. *The codomain of the `choose` function is a subset of $\mathbb{Z}$.*

Lemma BB3. *The integer-exponent of an integer is an integer.*

Lemma BB4. *The binomial `expansion` (defined in the `binomial` book) is a list of naturals.*

Lemma BB5. *For any list of naturals, its summation under the `sumlist` function is a natural.*

Now we get to the crux of the argument. Essentially, we will show that if $\alpha < 1$ is a rational with numerator p and denominator q, then $\alpha = p/q \leq p/(p+1)$ and thus $\alpha^p \leq p^p/(p+1)^p$. By the binomial theorem, $(p+1)^p \geq 2p^p$, thus $\alpha^p \leq 1/2$, allowing us to reduce to the argument from step 1. Formally speaking, we accomplish this through the following sequence of lemmas.

Lemma BB6. *For all $n \in \mathbb{N}_+$, $2n^n \leq (1+n)^n$.*

Proof. Follows from the binomial theorem because $(1+n)^n \leq 1 + \ldots + nn^{n-1} + n^n$. □

Now we prove a "helper lemma", similar to what we did in prior proof strategies but this time for the goal of "squeezing" α^n below $1/2$.

Lemma BB7. *For all $\alpha < 1$ in $\mathbb{Q}_+$, $\alpha^{numerator(\alpha)} \leq 1/2$.*

Proof. Let $x/y = \alpha$. By Lemma BB6, $2y^y \leq (1+y)^y$. By Lemma BA7, $(y^y)/(1+y)^y < (y^y)/(2y^y)$. But since for all $a,b,c \in \mathbb{Q}_+, a^b/c^b = (a/c)^b$ and $a/2a = 1/2$, it immediately follows that $(y/(1+y))^y < 1/2$. By Lemma BB1, $\alpha \leq \alpha/(1+\alpha)$. Combining this with the (non-strict) monotinicity of the exponent, clearly $\alpha^y \leq (\alpha/(1+\alpha))^y < 1/2$. The rest follows from Lemma BA8 (or the equivalent result, in the case of the manual proof). □

3.3 Summary and Closing Thoughts

In this Section we provided two rational proof strategies for Theorem 1 – the "ceiling proof" and the "binomial proof" – both of which we implemented using ACL2s. Since the rationals are dense in the reals, these proof strategies equally apply to the real numbers. The ceiling proof, which is the one we used in our prior work [4], involved first proving an intermediary lemma about the ceiling function. Specifically, assuming $0 < \alpha < 1$ and setting $k = \lceil \alpha/(1-\alpha) \rceil$ and $\delta = \lceil k\alpha^k/\varepsilon \rceil$, we proved that $\forall n \geq \delta$, $\alpha^n \leq f_\alpha(n)$. Since $f_\alpha(n) \leq \varepsilon$, the result directly followed. However, to prove this we first had to establish many arithmetic lemmas about the ceiling function, so although straightforward on paper, the proof was comparatively arduous in ACL2s.

For the "binomial proof", we broke the problem into two steps, first showing that if $0 < \alpha \leq 1/2$ then $\lim_{n\to\infty} \alpha^n = 0$, and then for any $\alpha \in (1/2, 1)$, constructing a δ such that $n > \delta \implies \alpha^n \leq 1/2$. For the first step, we showed two different approaches, one manual and the other semi-automatic. What made the second approach semi-automatic was that we used the termination analysis capabilities of ACL2s to find the "δ" for our ε/δ proof automatically. However, we still had to prove that this δ satisfied Definition 1. For the second step, we used the binomial theorem to show that for all positive integers p, $2p^p \leq (p+1)^p$ and therefore, if $\alpha = p/q \leq p/(p+1)$ then $\alpha^p \leq (p/(p+1))^p \leq 1/2$. Overall, both versions of the binomial proof were considerably simpler (in terms of lines of code) than the ceiling proof in ACL2s, and the comparison is fair given that both strategies required importing preexisting books (refer to Table 1).

Next, we return to the RTO, and show how any proof of Theorem 1 allows us to characterize the asymptotic behaviors of the srtt and rttvar. We also discuss the implications of these results for the rto.

4 Analysis of RTO Calculation

Recall from Eqn 1 that the rto is defined over the rttvar, srtt, and RTT sample S; the rttvar is defined over the S and the prior rttvar and srtt; and the srtt is defined over the S and prior srtt. Thus, going from the inside out, we begin by characterizing the srtt; then we use that analysis to aid our characterization of the rttvar; and finally we bring it all together to analyze the rto.

All of our work will be done under the "steady-state" assumption defined below. The purpose of this assumption is to define what it means for the network to exhibit bounded amounts of oscillation. Note however that this assumption does not limit the *rate* of oscillation in the sample values, for example, it does not require that the samples be drawn from the image of some Lipschitz continuous function.

Definition 2 (c/r Steady State)**.** *Let $c, r > 0$ be rationals and suppose that $\mathsf{S}_i, \mathsf{S}_{i+1}, \ldots, \mathsf{S}_{i+n} \in [c-r, c+r]$. Then we refer to the samples S_j for $j = i, \ldots, i+n$ as being in a c/r* steady-state.

For example, if $\mathsf{S}_i, \mathsf{S}_{i+1}, \ldots, \mathsf{S}_{i+n}$ are drawn from the uniform distribution over $[12.3, 75]$ms, then they are in a $43.65/31.35$ steady-state. Since every finite set achieves both a minimum and a maximum, all finite sequences of samples are technically speaking steady-state sequences, however, the same cannot be said for infinite sequences, such as the infinite sequence $\mathsf{S}_j = 12.3 + 2j$ for $j \in \mathbb{N}$.

Without loss of generality, for the rest of this section we will assume samples $\mathsf{S}_i, \mathsf{S}_{i+1}, \ldots, \mathsf{S}_{i+n}$ are in a c/r steady-state. We say WLOG because, we will consider both $n \in \mathbb{N}$ and the limit as $n \to \infty$. We begin by analyzing the srtt. Recall, $\mathsf{srtt}_i = \mathsf{S}_i$ if $i = 1$ else $(1-\alpha)\mathsf{srtt}_{i-1} + \alpha\mathsf{S}_i$. Since $\mathsf{S}_i \in [c-r, c+r]$:

$$(1-\alpha)\mathsf{srtt}_{i-1} + \alpha(c-r) \leq \mathsf{srtt}_i \leq (1-\alpha)\mathsf{srtt}_{i-1} + \alpha(c+r) \tag{10}$$

Note that both bounds in Eqn. 10 have the shape $(1-\alpha)\mathsf{srtt}_{i-1} + \alpha C$ for some constant C. Recursing on this shape, we get the following.

Lemma 1. *Let $C \in \mathbb{Q}_+$ and suppose that for all natural $0 \leq k \leq n$, we have $f(k) = (1-\alpha)\mathsf{srtt}_{i-1} + \alpha C$. Then the following holds for all $0 \leq k \leq n$.*

$$\begin{aligned} f(k) &= (1-\alpha)^{k+1}\mathsf{srtt}_{i-1} + \big(\sum_{j=0}^{k}(1-\alpha)^j\alpha\big)C \\ &= (1-\alpha)^{k+1}\mathsf{srtt}_{i-1} + \big((\alpha-1)(1-\alpha)^k + 1\big)C \end{aligned} \tag{11}$$

Applying Lemma 1 to Eqn. 10 we get the following.

Theorem 2. *Suppose $\mathsf{S}_i, \mathsf{S}_{i+1}, \ldots, \mathsf{S}_{i+n}$ are in a c/r steady-state. Then $L \leq \mathsf{srtt}_{i+n} \leq H$ where ...*

$$\begin{aligned} L &= (1-\alpha)^{n+1}\mathsf{srtt}_{i-1} + \big(1-(1-\alpha)^{n+1}\big)(c-r), \text{ and} \\ H &= (1-\alpha)^{n+1}\mathsf{srtt}_{i-1} + \big(1-(1-\alpha)^{n+1}\big)(c+r) \end{aligned} \tag{12}$$

In ACL2s, our proof strategy goes as follows. First we derive the closed form for $\sum_{j=0}^{k}(1-\alpha)^j\alpha$ and use it to rewrite srtt_{i+n} under the assumption that $\mathsf{S}_i = \mathsf{S}_{i+1} = \ldots = \mathsf{S}_{i+n}$. Then we show that in the c/r steady-state scenario, the lower bound L on the srtt_{i+n} is the value srtt_{i+n} would take if all the samples equaled $c-r$, and likewise the upper bound H is the value srtt_{i+n} would take if all the samples equaled $c+r$. Finally, we simplify and get the desired result. Finally, we look at the asymptotic case.

Theorem 3. $\lim_{n\to\infty} L = c-r$ *and* $\lim_{n\to\infty} H = c+r$.

Proof. For simplicity consider: $f(n) = (1-\alpha)^{n+1}\mathsf{srtt}_{i-1} + \big((\alpha-1)(1-\alpha)^n + 1\big)C$. Since $0 < \alpha < 1$, we know $0 < 1-\alpha < 1$, so by Theorem 1, $(1-a)^{n+1} \to 0$, and likewise for $(1-\alpha)^n$. This leaves only the term C. Thus $\lim_{n\to\infty} f(n) = C$. Since L is just $f(n)$ with $C = c-r$ and H is just $f(n)$ with $C = c+r$, the result immediately follows. □

Next, we consider the rttvar calculation. Recall that $\mathsf{rttvar}_i = \mathsf{S}_i/2$ if $i = 1$ else $(1-\beta)\mathsf{rttvar}_{i-1} + \beta|\mathsf{srtt}_{i-1} - \mathsf{S}_i|$, where $\beta < 1$ is constant in $\mathbb{Q}_+$. To simplify this equation, we will consider the case where $|\mathsf{srtt}_{i-1} - \mathsf{S}_i|$ is upper-bounded by some constant Δ. (Then we will show how to derive such a Δ).

Lemma 2. *Suppose $\mathsf{S}_i, \mathsf{S}_{i+1}, \ldots, \mathsf{S}_{i+n}$ are in a c/r steady-state, and $\Delta > 0$ upper-bounds $|\mathsf{srtt}_{j-1} - \mathsf{S}_j|$ for each $j = i, i+1, \ldots, i+n$. Then:*

$$\mathsf{rttvar}_{i+n} \leq (1-\beta)^{n+1}\mathsf{rttvar}_{i-1} + (1-(1-\beta)^{n+1})\Delta \tag{13}$$

Now we want to derive a bound on $|\mathsf{srtt}_{j-1} - \mathsf{S}_j|$. Note that this bound is at most:

$$\max\{|L-(c+r)|, |H-(c-r)|\} \tag{14}$$

If the larger of the two options is $|H-(c-r)|$ we derive the following.

$$\begin{aligned}\Delta &= |(1-\alpha)^{j+1}\mathsf{srtt}_{i-1} + (1-(1-\alpha)^{j+1})(c+r) - (c-r)| \\ &= |(1-\alpha)^{j+1}\mathsf{srtt}_{i-1} + (c+r) - (1-\alpha)^{j+1}(c+r) - c + r| \\ &= |(1-\alpha)^{j+1}\mathsf{srtt}_{i-1} + 2r - (1-\alpha)^{j+1}(c+r)|\end{aligned} \tag{15}$$

By Theorem 1, clearly $\lim_{j\to\infty}\Delta = 2r$. On the other hand, if the larger option is $|L-(c+r)|$, we derive $\Delta = |(1-\alpha)^{j+1}\mathsf{srtt}_{i-1} - 2r + (1-\alpha)^{j+1}(c-r)|$ which asymptotes at $|-2r| = 2r$. So either way, as j grows $\to\infty$, Δ converges to $2r$. Now suppose $\Delta = 2r$. Then the upper bound on srtt_{i+n} from Eqn 13 is $(1-\beta)^{n+1}\mathsf{rttvar}_{i-1} + (1-(1-\beta)^{n+1})2r$. Which gives us the following.

Theorem 4. *Suppose $\mathsf{S}_i, \mathsf{S}_{i+1}, \ldots, \mathsf{S}_{i+n}$ are in a c/r steady-state. Then there exists an upper bound on rttvar_{i+n} which, as $n\to\infty$, converges to $2r$.*

Proof. Follows from Theorem 1 because $\lim_{n\to\infty}(1-\beta)^{n+1}\mathsf{rttvar}_{i-1} = \lim_{n\to\infty}(1-\beta)^{n+1}2r = 0$. □

Finally, we turn our attention to the rto calculation. Recall, $\mathsf{rto}_i = \mathsf{srtt}_i + \max(G, 4\cdot\mathsf{rttvar}_i)$ for some constant G. On the one hand, if the rttvar is consistently very small (less than 1/4 of G) then clearly the rto is bounded by $[L+G, H+G]$. In this case, if $G > 2r$, we are assured that timeouts will never happen. But what if G is small relative to the radius of the steady-state interval? If $G < 2r$ and the rttvar can achieve a value $\leq G$ then a timeout can occur. And in fact, we can easily construct a scenario where exactly this happens infinitely many times.

For the pathological scenario, suppose that every 100th sample equals 75, while all the rest equal 60. Clearly the samples are in a 67.5/7.5 steady-state. At the spikes (where $\mathsf{S}_{i+100n} = 75$), $\mathsf{srtt} \approx 61.88, \mathsf{rttvar} \approx 3.75$, and $\mathsf{rto} \approx 61$. Since $61 < 75$, a timeout occurs. There are infinitely many "spikes" where timeouts occur. However, when we simulate a scenario where the samples are uniformly random over $[c-r, c+r]$, little to no timeouts occur. Both scenarios are illustrated below in Fig. 2.[8]

The RTO calculation is specified in RFC6298 [9] which says the constant G should be set to the "clock granularity" of the sender, in seconds. In the interest of avoiding excessive timeouts, a protocol implementer might want to consider adding the additional criteria that G should exceed 1/2 the diameter of the maximally large sample interval that they consider to be "stable". Depending on the nature and context of the protocol, this could be either a fixed or dynamic value. However, caution should be taken on the other hand to ensure G is not too large, since timeouts *should* occur when there really is congestion on the network, in order to avoid congestion collapse. Also, a dynamic value of G could exacerbate the risk of choosing too large of a timeout value, and might even be vulnerable to targeted manipulation.

As we alluded when introducing Def 2, one interesting direction for future research is to investigate minimal, sufficient analytic conditions to ensure timeouts do not occur. For example, it might be sufficient to require the samples be drawn from the image of a Lipschitz continuous function, and then to require some relationship between the Lipschitz bound, the choice of β, and the radius r. Although this is purely speculative, it is certainly the case that the derivative of the samples must be taken into account when analyzing the magnitude of the rttvar, providing multiple interesting directions for future research.

[8] Adapted from Fig. 3 of [4], first published in volume 14067, page 56, 2023, by Springer Nature.

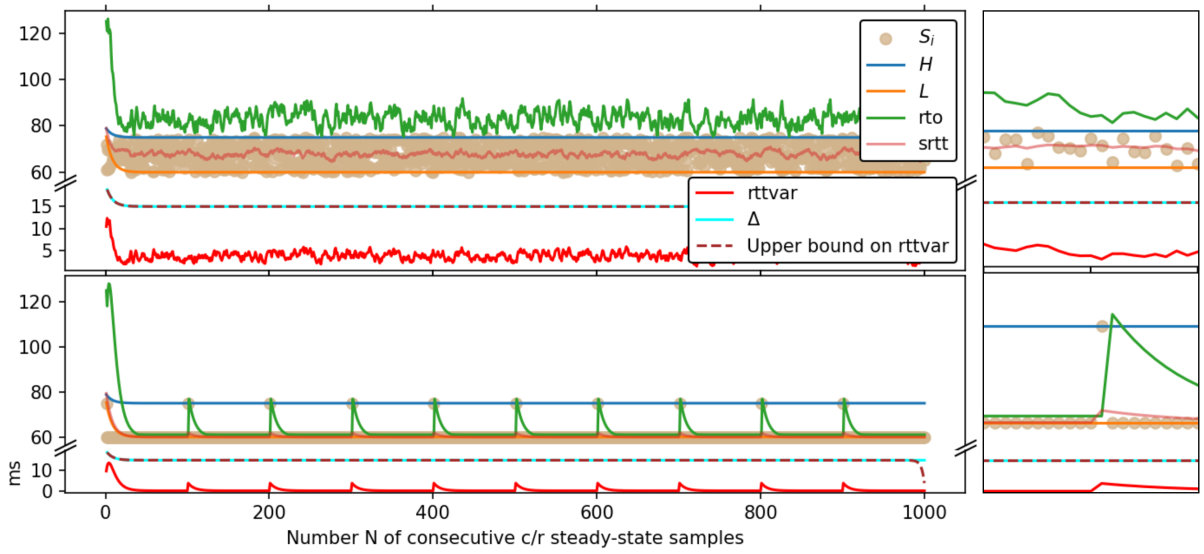


Figure 2: On the left are two 67.5/7.5 steady-state scenarios. On top the samples are drawn from the uniform distribution over the bounds, and timeouts rarely, if ever, occur. In the bottom (pathological) scenario, every 100$^{\text{th}}$ sample equals $c+r=75$ while the rest equal $c-r=60$, and at each "spike", a timeout occurs. There are infinitely many spikes, and one is shown on the right ($n=[350,450]$).

5 Discussion

In this work we considered multiple approaches to proving $\forall\alpha\in[0,1)::\lim_{n\to\infty}\alpha^n=0$. Our first was in ACL2(r) and resembled the obvious pen-and-paper proof, but could not be imported into an ACL2 environment. Our second and third were in ACL2s and required proving some arithmetic lemmas. Of these, the semi-automated version of the third is the most stylistically aligned with ACL2s because it takes advantage of automated termination analysis. The ACL2(r) proof has the lowest character count of all the proofs, and certifies the most quickly, but imports the most books and is not portable to ACL2. Considering books and portability, the semi-automated binomial proof is probably the best (see Tab. 1).

While working on the third approach we encountered an inconvenience in ACL2s: in ACL2s, `definecs` support `function-` and `body-contract-hints`, but `property` definitions do not. We found two ways to address this. The first was to redefine the problematic property as a decision procedure in a

Proof	LoC	Chars	Props/Thms	Functions	Books	Cert Time (s)
Real	161	4,224	17	1	5	0.58
Ceiling	408	16,103	20	3	0	64.17
Binomial (M)	154	5,652	22	1	2	2.54
Binomial (SA)	122	5,402	22	2	1	3.84

Table 1: Proof comparison. (M) refers to "manual" while (SA) refers to "semi-automatic". Lines of code and character count are computed without comments or empty lines, however, the proofs are not styled identically. Props/Thms counts instances of `property` and `defthm`, while Functions counts `definecs`, `definecds`, and `defuns`. Certification time is measured on a 16GB M1 Macbook Air.

`definec` with appropriate hints, and then write a new property saying that on all inputs, the procedure returns *true*. The second was simply to set `:check-contracts? nil`. To ameliorate this issue, we plan to add hints to `property` definitions in a future update to ACL2s.

While writing the ACL2s proofs, we took advantage of ACL2s features not available in ACL2 or ACL2(r). Most notably, we used termination analysis in the semi-automated binomial proof. There are also smaller ways that ACL2s was easier: type annotations improved the readability of our proofs, and contract-checking gave us some lemmas for free, including type-checking in the `xargs` to our `defun-sk`. In contrast, we had to manually prove contracts for d_α in our ACL2(r) proof, and it is harder to read than either ACL2s proof (with lengthier antecedents on `defthms`) because it lacks type annotations.

6 Conclusion

Recently, the Internet Engineering Task Force created a Usable Formal Methods Research Group, of which we are members, to integrate formal methods into the RFC drafting process. In many protocols, *performance* matters: we want to know how quickly the protocol achieves a desired outcome under load. Usually performance is studied using simulations or measurements, which can give a sense of how protocols behave "in the wild". But what about how protocols behave in the worst case? What if the worst case never happens in the measured environments or simulations? For this, we need a way to *prove* performance bounds, namely, formal methods. Meanwhile, real analysis provides a convenient framework with which to ask and answer questions about performance bounds in the long run. In our prior work [4], we used formal methods with real analysis to prove useful bounds on the internal variables of the RTO calculation. But we also ran into hurdles. We could not use the most obvious proof, which requires real numbers, because ACL2s only supports rationals. When we came up with an alternative approach (the "ceiling proof") it required us to convince ACL2s of numerous arithmetic lemmas. Only post-publication did we find a simpler solution, based on the binomial theorem.

As relatively novice users of ACL2s[9], our work leads us to identify three areas where we feel the ACL2 ecosystem could be improved to support work such as ours. First, ACL2 (and ACl2s in particular) could benefit from a richer, more comprehensively documented, and more easily searchable library of purely mathematical theorems, relating to the ceiling, floor, exponent, and logarithm, as well as metric spaces and limits. Searching for proofs is difficult enough, and ACL2 does not come with any kind of semantic proof search tool. And often, even when the desired theorems exist in the ACL2 books, they are unmentioned in the documentation. For example, the documentation on "arithmetic" does not mention the RTL books, and neither does the documentation on "math". Moreover, the rewrite rules from different libraries may conflict, so even if you find the desired theorems, importing them into a singular environment may be non-trivial. It would also be useful to have more mathematics formalized in ACL2s, so as to avoid additional proof obligations (for function contracts and termination) when using imported books. Second, ACL2(r) could benefit from the addition of the generic exponent and logarithm. This could be done using the translational Lemma given in §2. Third, and most importantly, though ACL2(r) and ACL2 have incompatible theories, it is nevertheless true that certain kinds of theorems over the reals should hold over the rationals, because the rationals are dense in the reals. It would useful to have a kind of "bridge" between ACL2(r) and ACL2, by which the user could justify that a given theorem, if true over the reals, must also hold over the rationals; prove the theorem in ACL2(r); and then import the theorem, using its "justification", into ACL2. Hopefully our experience provides insight for future work in both protocol analysis and extending the ACL2 ecosystem.

[9] Excluding the second author.

Acknowledgments. The first author would like to thank Ruben Gamboa for providing technical support in ACL2(r) and suggesting Lemma R1, and Ankit Kumar for providing technical support in ACL2s.

References

[1] Harsh Chamarthi, Peter C. Dillinger, Panagiotis Manolios & Daron Vroon (2011): *The "ACL2" Sedan Theorem Proving System*. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, doi:10.1007/978-3-642-19835-9_27.

[2] Peter C. Dillinger, Panagiotis Manolios, Daron Vroon & J Strother Moore (2007): *ACL2s: "The ACL2 Sedan"*. In: *International Conference on Software Engineering (ICSE)*, doi:10.1109/ICSECOMPANION.2007.14.

[3] Ruben A Gamboa & Matt Kaufmann (2001): *Nonstandard analysis in ACL2*. *Journal of automated reasoning* 27, pp. 323–351, doi:10.1023/A:1011908113514.

[4] Max von Hippel, Kenneth L. McMillan, Cristina Nita-Rotaru & Lenore Zuck (2023): *A Formal Analysis of Karn's Algorithm*. In: *2023 International Conference on NETworked sYStems (NETYS)*, Springer, doi:10.1007/978-3-031-37765-5_4.

[5] Phil Karn & Craig Partridge (1987): *Improving round-trip time estimates in reliable transport protocols*. *ACM SIGCOMM Computer Communication Review* 17(5), pp. 2–7, doi:10.1145/55483.55484.

[6] Ankit Kumar & Panagiotis Manolios (2021): *Mathematical Programming Modulo Strings*. In: *2021 Formal Methods in Computer Aided Design (FMCAD)*, IEEE, pp. 261–270, doi:10.34727/2021/isbn.978-3-85448-046-4_36.

[7] Panagiotis Manolios & Daron Vroon (2006): *Termination analysis with calling context graphs*. In: *Computer Aided Verification: 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006. Proceedings 18*, Springer, pp. 401–414, doi:10.1007/11817963_36.

[8] Oded Padon, Kenneth L McMillan, Aurojit Panda, Mooly Sagiv & Sharon Shoham (2016): *Ivy: safety verification by interactive generalization*. In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 614–630, doi:10.1145/2908080.2908118.

[9] V. Paxson, M. Allman, J. Chu & M. Sargent (2011): *Computing TCP's Retransmission Timer*. `https://datatracker.ietf.org/doc/html/rfc6298`. Accessed 15 June 2023.

[10] Brian M. Scott (https://math.stackexchange.com/users/12042/brian-m scott): *Nested Division in the Ceiling Function*. Mathematics Stack Exchange. Available at `https://math.stackexchange.com/q/233684`. (version: 2012-11-09).

Advances in ACL2 Proof Debugging Tools*

Matt Kaufmann and J Strother Moore
Department of Computer Science, The University of Texas at Austin, Austin, TX, USA (retired)
{kaufmann,moore}@cs.utexas.edu

The experience of an ACL2 user generally includes many failed proof attempts. A key to successful use of the ACL2 prover is the effective use of tools to debug those failures. We focus on changes made after ACL2 Version 8.5: the improved break-rewrite utility and the new utility, `with-brr-data`.

1 Introduction

The key technique for debugging ACL2 proofs is known as *the method*. [1] Briefly put, it is to look at *checkpoints* that the prover cannot further simplify, usually to get ideas for controlling rewriting by introducing new rewrite rules or by enabling or disabling existing ones. However, there are many proof debugging tools that can be helpful when using the method; brief summaries may be found in the list of debugging topics (for proofs and otherwise) in the ACL2+books manual [3]. Among these are a number of popular tools, including `accumulated-persistence` and the proof-builder.

However, this paper focuses on the following two tools: `With-brr-data` and the break-rewrite utility. These are tools that directly track the ACL2 rewriter. The former was introduced after the release of ACL2 Version 8.5 in July, 2022. The latter was introduced in the early 1990s (Version 1.3) but was significantly improved after the Version 8.5 release. The sections below deal with each of these, first at the user level and then with implementation-level discussions. The user-level discussions start with the new tool, `with-brr-data`, followed by a section on break-rewrite (including its new "near misses" capability) and then a section showing their use together. The implementation-level sections start with background material on a key enabling device, wormholes, followed by a discussion of break-rewrite implementation, after which we discuss the implementation of `with-brr-data` and how it takes advantage of the break-rewrite implementation. Next we discuss how to change the functionality of `with-brr-data` with attachments. We wrap up with a conclusion. We assume some familiarity with rewriting in ACL2.

The examples in this paper are available in supporting materials for this paper; see community books file `workshops/2023/kaufmann-moore/README`. Additional examples may be found in `demos/brr-test-input.lsp`, which generates output found in `demos/brr-test-log.txt`, and in `system/tests/brr-data-input.lsp`, which generates output found in `system/tests/brr-data-log.txt`.

Terminology. An *application A* of a rewrite rule *R* is the process of replacing a term, the *target*, by suitably rewriting the right-hand side of *R* to obtain the *result* of *A*. This process includes not only rewriting the right-hand side of *R* but also relieving the hypotheses of *R*. For this purpose, the equality $(fx_1 \ldots x_k) = b$ representing a definition is viewed as a rewrite rule whose left- and right-hand sides are those of the equality. A term t_1 is said to *contain* t_2 if t_2 occurs as a subterm of t_1. A rewrite rule application *introduces tm as a subterm* if the result, but not the target, contains *tm*. For a given application *A* of a rewrite rule *R*, a *subsidiary application* is a rewrite rule application that takes place after *A* begins

*Released under Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).

[1] The online version of this paper provides underlined links like this to documentation [3] topics.

A. Coglio, S. Swords (Eds.): 18th ACL2 Workshop (ACL2-2023)
EPTCS 393, 2023, pp. 67–81, doi:10.4204/EPTCS.393.7

but before *A* completes; that is, it takes place during the process of rewriting the right-hand side of *R*. Thus when *A* is represented by a frame displayed by the utility cw-gstack or (similarly) the break-rewrite path command, all rewrite rule applications represented below that frame are subsidiary to *A*. Finally, a *top-level* operation of the simplifier is any operation performed by the simplifier that is not made while attempting to relieve a hypothesis.

2 User-level introduction to with-brr-data

ACL2 users sometimes find a surprising term in a checkpoint. This section summarizes how the with-brr-data utility may be used to see how rewrite rules were applied to produce a given term during a proof attempt. These rules might well include at least one rule that hasn't occurred to the user, perhaps because it was introduced by including someone else's book. See also the with-brr-data documentation for more details and examples.

The behavior of associated queries is perhaps best explained with the examples below, but we start here with a summary. If *F* is a form that invokes the prover, then `(with-brr-data` *F*`)` saves prover data that can be queried later. A key query is of the form `(cw-gstack-for-subterm` *tm*`)` where *tm* is a term; variants are described near the end of this section. That query searches for the first top-level rewrite rule application, *A* — the *product* of the search — that introduced *tm* as a subterm. If *A* is found, then a stack *S* is displayed as with cw-gstack or (similarly) the break-rewrite path command (hence using terms in translated form). *S* represents top-level operations down to a frame *F* that represents *A*, and *S* maximally extends past *F* such that every rule application represented below *F* (which is necessarily subsidiary to *A*) is *suitable*: its result contains *tm* as a subterm. After the stack is printed, the result of its final rewrite rule application is printed. If the stack extends beyond *A* then the result of *A* is also printed.

Next we give two examples that illustrate the description above. Let's start with the simpler one.

```
(include-book "std/lists/rev" :dir :system)
(with-brr-data
 (thm (implies (and (natp n)
                    (< n (len x)))
               (equal (nth n (revappend x y))
                      (nth n (reverse x))))
      :hints ; The second example shows what happens when we remove :hints.
      (("Goal" :do-not '(preprocess)))))
```

We see the following checkpoint at the top level.

```
(IMPLIES (AND (INTEGERP N)
              (<= 0 N)
              (< N (LEN X))
              (NOT (STRINGP X)))
         (EQUAL (NTH N (APPEND (REV X) Y))
                (NTH N (REV X))))
```

At this point we might reasonably ask: How did rewriting produce `(REV X)`? Note that `REV` is not a built-in function; it must have been defined in an included book. The following log, explained below, shows how a query using `cw-gstack-for-subterm` can provide an answer.

```
  ACL2 !>(cw-gstack-for-subterm (REV X))
  1. Simplifying the clause
       ((IMPLIES (IF (NATP N) (< N (LEN X)) 'NIL)
                 (EQUAL (NTH N (REVAPPEND X Y))
```

```
                       (NTH N (REVERSE X)))))
  2. Rewriting (to simplify) the atom of the first literal,
       (IMPLIES (IF (NATP N) (< N (LEN X)) 'NIL)
                (EQUAL (NTH N (REVAPPEND X Y))
                       (NTH N (REVERSE X)))),
  3. Rewriting (to simplify) the second argument,
       (EQUAL (NTH N (REVAPPEND X Y))
              (NTH N (REVERSE X))),
  4. Rewriting (to simplify) the first argument,
       (NTH N (REVAPPEND X Y)),
  5. Rewriting (to simplify) the second argument,
       (REVAPPEND X Y),
  6. Attempting to apply (:REWRITE REVAPPEND-REMOVAL) to
       (REVAPPEND X Y)
  The resulting (translated) term is
    (BINARY-APPEND (REV X) Y).
  ACL2 !>
```

The use of `with-brr-data` above caused data to be collected that we can query as shown in the log above. Frame 1 shows the initial clause (list of literals, implicitly disjoined), which in this case is a list containing just the initial translated goal. As we look down the stack we see how the process of simplification moved from there to frame 6, which is what the search was seeking: the first rewrite rule application *A* that introduced `(REV X)` as a subterm; as noted above, we will call *A* the *product of the search*. The stack stops there because no rewrite rules were applied to the right-hand side after applying `REVAPPEND-REMOVAL`. The log concludes with the result of *A*.

The second example modifies the first by removing the `:hints`. That produces the following proof output, for example by using `:pso`.

```
By the simple :definition NATP and the simple :rewrite rule REVAPPEND-REMOVAL
we reduce the conjecture to

Goal'
(IMPLIES (AND (INTEGERP N)
              (<= 0 N)
              (< N (LEN X)))
         (EQUAL (NTH N (APPEND (REV X) Y))
                (NTH N (REVERSE X)))).
```

This goal further simplified to produce the same checkpoint as the first example. But the word "simple" in the log above indicates that simplification was performed with the lightweight "preprocess" simplifier. Like break-rewrite, `with-brr-data` does not store data from the preprocess simplifier. In particular, nothing was stored while generating the goal above. Without such data, it is impossible to track the source of the subterm `(REV X)` occurring in `Goal'`, since it has already been put into that goal with preprocessing and we are looking for rewrite rule applications that *introduce* `(REV X)` as a subterm.

However, simplification of `Goal'` introduced a new occurrence of `(REV X)`, as seen in the following log; explanation follows.

```
  ACL2 !>(cw-gstack-for-subterm (REV X))
  1. Simplifying the clause
       ((NOT (INTEGERP N))
        (< N '0)
        (NOT (< N (LEN X)))
        (EQUAL (NTH N (BINARY-APPEND (REV X) Y))
               (NTH N (REVERSE X))))
```

```
2. Rewriting (to simplify) the atom of the fourth literal,
     (EQUAL (NTH N (BINARY-APPEND (REV X) Y))
            (NTH N (REVERSE X))),
3. Rewriting (to simplify) the second argument,
     (NTH N (REVERSE X)),
4. Rewriting (to simplify) the second argument,
     (REVERSE X),
5. Attempting to apply (:DEFINITION REVERSE) to
     (REVERSE X)
6. Rewriting (to simplify) the body,
     (IF (STRINGP X)
         (COERCE (REVAPPEND (COERCE X 'LIST) 'NIL)
                 'STRING)
       (REVAPPEND X 'NIL)),
   under the substitution
     X : X
7. Rewriting (to simplify) the third argument,
     (REVAPPEND X 'NIL),
   under the substitution
     X : X
8. Attempting to apply (:REWRITE REVAPPEND-REMOVAL) to
     (REVAPPEND X 'NIL)
9. Rewriting (to simplify) the rhs of the conclusion,
     (BINARY-APPEND (REV X) Y),
   under the substitution
     Y : 'NIL
     X : X
10. Attempting to apply (:REWRITE APPEND-ATOM-UNDER-LIST-EQUIV) to
     (BINARY-APPEND (REV X) 'NIL)
The resulting (translated) term is
  (REV X).
Note: The first lemma application above that provides a suitable result
is at frame 5, and that result is
  (IF (STRINGP X)
      (COERCE (REV (COERCE X 'LIST)) 'STRING)
    (REV X)).
ACL2 !>
```

The clause in frame 1 corresponds to the goal above, `Goal'`; thus this stack is from the process of simplifying that goal. Frames 2 and 3 show that we are dealing with the second argument of the call of `EQUAL`, which unlike the first argument did not already have a subterm of `(REV X)`. Frame 5 shows the product of the search: the first rewrite rule application that introduced `(REV X)` as a subterm, whose result is shown in the Note printed at the end about "a suitable result". For the rest of the stack after frame 5, `(REV X)` is a subterm of the result of each rewrite rule application. Just after the stack is printed (and before the Note at the end, which is about frame 5), we see the result of the rule application of the final frame, frame 10, which happens to be `(REV X)` itself.

The second example above illustrates why the stack is extended past the product of the search using rule applications whose result contains the subterm of the query, in this case `(REV X)`. If the display had ended at frame 5, we would not have seen the rule most directly responsible for introducing `(REV X)` as a subterm — `REVAPPEND-REMOVAL`, applied at frame 8 – even though `(REV X)` occurs in the result from frame 5 (as noted at the end of the log above).

For more examples, see the community books file, `system/tests/brr-data-input.lsp` and corresponding log `brr-data-log.txt` in that directory.

The following related query capabilities are also supported.

- The query `(cw-gstack-for-term` *tm*`)` differs from `(cw-gstack-for-subterm` *tm*`)` only in that it searches results for *tm* itself, rather than for terms containing *tm* as a subterm.
- The queries `(cw-gstack-for-subterm*` *tm*`)` and `(cw-gstack-for-term*` *tm*`)` are *iterative* versions of their counterparts without the '`*`' suffix, in that that they query for additional results. Each successive search in the scope of these queries ignores all rule applications that are subsidiary to the products of previous searches.
- All of these utilities can take an argument of the following form (as permitted for `:expand` hints): `(:free (`v_1 ... v_k`)` *tm*`)`, where the v_i are variables and *tm* is a term. In this case, the search is for any instance of *tm* obtained by substituting for the v_i. Note however that once the product of a search is found, the corresponding instance of *tm* is used for finding a maximal stack extension.

We conclude this section with three remarks.

1. Rules are monitored within the scope of `with-brr-data` as though `:brr t` has been executed.
2. `With-brr-data` does not collect any data for the near-miss breaks discussed in the next section.
3. In ACL2(p), `with-brr-data` is disallowed when waterfall-parallelism is enabled, as that would interfere with the sequential nature of data collection (which relies on the timing for collection of subsidiary applications).

3 User-level introduction to break-rewrite

Break-rewrite was originally designed to help answer the question "why did the attempt to apply a certain lemma fail?" It was modeled on Nqthm's `break-lemma` [1, pp. 257–264].[2] For example, suppose the user has proved these two rules,

```
(defthm p-rule  (implies (q x) (p (f x y))))
(defthm q-rule1 (implies (r x) (q x)))
```

and then tries `(thm (implies (r v) (p (f u v))))`. The proof fails. But the user expected the two rules to be used to prove the theorem and so responds with

```
(brr t)                           ; turn on break-rewrite
(monitor 'p-rule  t)              ; unconditionally break when p-rule is matched
(monitor 'q-rule1 t)              ; unconditionally break when q-rule1 is matched
(thm (implies (r v) (p (f u v)))) ; try thm again
```

This time there are interactive breaks. The keyword commands below are the user's responses to the breaks. We have indented the depth 2 break for clarity; ACL2 does not indent breaks, to save space on the line.

```
(1 Breaking (:REWRITE P-RULE) on (P (F U V)):
1 ACL2 >:eval

      (2 Breaking (:REWRITE Q-RULE1) on (Q U):
      2 ACL2 >:eval

      2x (:REWRITE Q-RULE1) failed because :HYP 1 rewrote to (R U).
```

[2] Nqthm's `break-lemma` is also described in [2, pp. 305–311].

```
2 ACL2 >:type-alist

Decoded type-alist:
-----
Terms with type (TS-COMPLEMENT *TS-NIL*):
(R V)

==========
Use (GET-BRR-LOCAL 'TYPE-ALIST STATE) to see actual type-alist.
2 ACL2 >:a!
Abort to ACL2 top-level.
```

We see that `q-rule1` failed because its first hypothesis rewrote to `(R U)` but the `:type-alist` shows `(R V)` as a given. We should be trying to prove `(thm (implies (r u) (p (f u v))))`.

Break-rewrite was helpful here because the user expected a certain rule to be tried and a target matching the rule's left-hand side was encountered by the rewriter, but something prevented the application. Typical failure reasons are that a hypothesis could not be relieved, a suitable free variable instantiation could not be found, or the rule would have performed a "heuristically unattractive" replacement.

But if the rewriter never sees a target that matches the rule, break-rewrite cannot help us — or at least it could not help us until the recent addition of *near-miss* break criteria.

We will illustrate a near-miss break with a lemma about `loop$` [4]. Near-miss break criteria are more general than this example might suggest. We elaborate at the end of this section. But four facts team up to make lemmas about `loop$` particularly prone to near-miss mismatches. (a) `Loop$`s create quoted `lambda` constants; (b) each `lambda` constant contains arbitrary ACL2 code, namely the `loop$` body; (c) the prover can rewrite `lambda` constants in slots of ilk `:FN`; but (d) matching requires identity on quoted constants. These facts often mean that lemmas about `loop$` fail to match because the `lambda` constants in the lemmas are not in "rewrite-normal form" because we typically do not write code in rewrite-normal form. ("Rewrite-normal form" is an informal notion. A term is in rewrite-normal form if it is not changed by rewriting. Of course, this really depends on the context in which the term occurs.)

Here is an example. Suppose `(nats n)` has been defined to return a list of natural numbers and `foo` is some function of one argument. Suppose the user wants to prove

```
(defthm thm-a
  (loop$ for e in (nats (foo a)) always (atom e)))
```

and has realized a more general lemma is needed:

```
(defthm lemma-a
  (loop$ for e in (nats n) always (atom e)))
```

This lemma is easily proved after including the standard `"projects/apply/top"` book. But when the user tries to prove `thm-a` after the proving `lemma-a` the proof fails. Monitoring `lemma-a` does not help: no break happens because no target term matching the lemma ever arises. What we need to do is see the targets that are near-misses.

Because our lemma involves `loop$` (and thus `lambda` constants) we install a near-miss monitor that means "break when a match fails due only to mismatching `lambda` constants."

```
(monitor 'lemma-a '(:lambda t))
```

Trying `thm-a` again produces an interactive break to which the user types the `:lhs` command.

```
(1 Breaking (:REWRITE LEMMA-A) on
(ALWAYS$ '(LAMBDA (LOOP$-IVAR) (IF (CONSP LOOP$-IVAR) 'NIL 'T))
         (NATS (FOO A))):

The pattern in this rule failed to match the target.  However, this
is considered a NEAR MISS under the break criteria,
(:CONDITION 'T :LAMBDA T), specified when this rule was monitored.
The following criterion is satisfied.

* :LHS matches :TARGET except at one or more quoted LAMBDA constants.

1 ACL2 >:lhs
(ALWAYS$ '(LAMBDA (LOOP$-IVAR) (ATOM LOOP$-IVAR))
         (NATS N))
```

The `:target` is shown in the break header. We see that the body of the quoted `lambda` constant in `:target` is `(IF (CONSP LOOP$-IVAR) 'NIL 'T)` but the body of the quoted `lambda` constant in `:lhs` is `(ATOM LOOP$-IVAR)`. The `lambda` constant in the theorem was rewritten; `(ATOM LOOP$-IVAR)` was expanded.

Problems of this sort can often be addressed in several ways, e.g., by disabling the rewriting of `lambda` objects (see rewrite-lambda-object), or disabling the functions in the body that the rewriter expanded. But, as in this case, those approaches often raise other issues. It is generally best to restate the lemma so that the body of the `lambda` is in rewrite-normal form. That is, restate `LEMMA-A` so that the body of the `loop$` is `(if (consp e) nil t)` instead of `(atom e)`. The experienced ACL2 user would not formulate a rewrite rule containing a non-recursive function like `atom` in its left-hand side but that is exactly what we did in our original formulation of `lemma-a`.

Other near-miss criteria are `:depth k` where `k` is a natural number and `:abstraction pat` where `pat` is a term. The former criterion triggers a break if the `:lhs` matches the target down to depth `k`. The latter triggers a break if the specified `pat` matches the target. All three criteria, `:lambda`, `:depth`, and `:abstraction`, are implemented the same way: the near-miss criterion gives rise to a *near-miss pattern* that is typically more general than the left-hand side, e.g., `:depth 2` applied to a left-hand side of `(f (g (h x) x))` generates the near-miss pattern (F (G GENSYM0 X)). We then try to instantiate the near-miss pattern to produce the target and if it succeeds we say that a near miss occurred (since the left-hand side failed to unify), and a corresponding message is printed. Note that each near-miss criterion is handled this way, and a message is printed for each that has occurred. This process is carried out by the function `brr-near-missp`. We hope to make `brr-near-missp` attachable so users can add additional ways to trigger near-miss breaks, but we have not done so yet. See `monitor` and `brr-near-missp` for details.

4 Using `with-brr-data` and break-rewrite together

Sometimes `with-brr-data` doesn't quite do the job by itself but is useful in concert with break-rewrite, as illustrated by the following example from the documentation for `with-brr-data`. This example does not exercise the near-miss feature of break-rewrite.

```
(with-brr-data (thm (equal (append x y) (append y x))))
```

After the proof attempt fails, the first checkpoint under induction is as follows.

```
(IMPLIES (AND (CONSP X)
```

```
            (EQUAL (APPEND (CDR X) Y)
                   (APPEND Y (CDR X))))
      (EQUAL (CONS (CAR X) (APPEND Y (CDR X)))
             (APPEND Y X)))
```

Even an experienced user might be surprised, at least initially, to see the second occurrence of `(APPEND Y (CDR X))`. The following log shows how an appropriate query can attempt to shed light on that.

```
ACL2 !>(cw-gstack-for-subterm (append y (cdr x)))
1. Simplifying the clause
     ((NOT (CONSP X))
      (NOT (EQUAL (BINARY-APPEND (CDR X) Y)
                  (BINARY-APPEND Y (CDR X))))
      (EQUAL (BINARY-APPEND X Y)
             (BINARY-APPEND Y X)))
2. Rewriting (to simplify) the atom of the third literal,
     (EQUAL (BINARY-APPEND X Y)
            (BINARY-APPEND Y X)),
3. Rewriting (to simplify) the first argument,
     (BINARY-APPEND X Y),
4. Attempting to apply (:DEFINITION BINARY-APPEND) to
     (BINARY-APPEND X Y)
The resulting (translated) term is
  (CONS (CAR X)
        (BINARY-APPEND Y (CDR X))).
ACL2 !>
```

This shows us that the term `(BINARY-APPEND Y (CDR X))`, which is the translated term corresponding to the user input of `(APPEND Y (CDR X))`, is produced from the definition of `BINARY-APPEND`. But how? We can monitor that definition to answer that question. Here is a log, abbreviated as shown.

```
ACL2 !>:monitor! binary-append (equal (brr@ :target) '(BINARY-APPEND X Y))
 T
ACL2 !>(thm (equal (append x y) (append y x)))

 [[.. Use :go to get past the start of induction. ..]]

*1 (the initial Goal, a key checkpoint) is pushed for proof by induction.

 [[.. elided ..]]

(1 Breaking (:DEFINITION BINARY-APPEND) on (BINARY-APPEND X Y):
1 ACL2 >:eval

1! (:DEFINITION BINARY-APPEND) produced
(CONS (CAR X) (BINARY-APPEND Y (CDR X))).

1 ACL2 >:type-alist

Decoded type-alist:
-----
Terms with type *TS-T*:
(EQUAL (APPEND (CDR X) Y)
       (APPEND Y (CDR X)))
-----
Terms with type *TS-CONS*:
X
```

```
==========
Use (GET-BRR-LOCAL 'TYPE-ALIST STATE) to see actual type-alist.
1 ACL2 >
```

We see that an equality — in this case, the induction hypothesis — has been applied to switch the arguments of the call of BINARY-APPEND that was created by applying its definition.

5 Brief introduction to wormholes

The implementations of with-brr-data and break-rewrite both depend on ACL2 wormholes. Here we provide a bit of relevant background on wormholes.

Every ACL2 function is a true mathematical function: equal inputs produce equal outputs. There are no side-effects. To interact with the user a function must take and return the ACL2 state so that inputs can be read and outputs printed.

Wormhole is an ACL2 function that takes a *wormhole name* and some other arguments, not including state, and always returns nil. So logically it is a trivial constant function. But when wormhole is called a new read-eval-print loop is started on a copy of the ACL2 state which also contains the inputs to the call of wormhole and an object, called the *wormhole status* associated with wormhole name as of the last time the wormhole was exited. While in this loop, forms can print information to the comment window, inspect and compute a new wormhole status, and (to a limited extent) modify the copy of the state available in the wormhole. But the first form "read" and executed by this loop is not one typed by the user but is provided in the call of wormhole. Thus the call of wormhole can inspect the available information and configure the status as appropriate and then either exit the loop (without ever prompting the user for anything) or stay in the loop and prompt the user for input. When the loop is exited for any reason the final status is saved in a secret location to be reinstated the next time that wormhole is entered. The rest of that copy of the state is discarded.

Thus, using a wormhole, you can accumulate into the status any data passed into the wormhole or obtained from state, you can print data, and you can interact with the user, but you cannot pass data from inside the wormhole out to the caller: the result is always nil and the ACL2 state remains unchanged.

Wormholes necessitate the "copying" of the ACL2 state and the saving of data outside of the ACL2 state. The copying is just an illusion. The state inside a wormhole is the "live state" but all allowed changes are tracked, including changes to state global variables, and when the wormhole exits, those changes are undone by Lisp's unwind protection mechanism. (This actually messed up the behavior of break-rewrite, e.g., by losing track of which lemmas are monitored, when the user invoked the theorem prover recursively from within a wormhole; but that problem has been fixed.) As for saving data outside the state, we use a raw Lisp association list to associate each wormhole name with its current status. When the wormhole is entered, that status object is assigned to a state global variable in the ACL2 state so that the status is visible to forms executing in the wormhole. When the wormhole is exited, that status is written back to the raw Lisp association list. This general scheme has been in effect for 30 years, but in working on break-rewrite recently we discovered some problems that necessitated clarifying the implementation of wormholes. The first step in that clarification is to introduce some terminology: the status of a wormhole stored in raw Lisp is called the *persistent* wormhole status ("whs"), while the status occasionally stored in the ACL2 state is called the *ephemeral* whs because it disappears and reappears. The problem we discovered with the old implementation of wormholes can best be understood by thinking of the persistent whs as a hard-to-access memory location and the ephemeral one as an easily accessed,

nearby cache. The problem was that our cache was not always coherent: some functions (or user commands) changed one without changing the other. See wormhole and wormhole-programming-tips for details.

Wormholes can be expensive to enter and exit, e.g., cleanup forms must be consed up upon state changes and evaluated upon exit, and forms executed within the wormhole must read, translated, and interpreted. So we provide a more efficient mechanism called wormhole-eval, which essentially takes a wormhole name and a lambda expression, binds the lambda formal to the persistent status of the named wormhole, evaluates the term, and stores the result back into the persistent status.

6 Implementation aspects for break-rewrite

The ACL2 rewriter does not modify state, so break-rewrite is implemented by writing to a wormhole named brr. The status of the brr wormhole is basically a state machine that records information about the rewriter's activities. The status is represented in an ACL2 defrec object.[3]

```
(defrec brr-status
  (entry-code (brr-monitored-runes . brr-gstack)
              . (brr-local-alist . brr-previous-status))
  t)
```

See wormhole for an explanation of entry-code. The next four components are the list of monitored runes and their break criteria, the rewriter's call stack, an alist binding variables passed in from the rewriter (and a few specific to the given break), and the previous brr-status. We think of a brr-status object as a stack: brr-monitored-runes, brr-gstack, and brr-local-alist characterize an active (still open) call of break-rewrite and brr-previous-status is the stack of calls leading to this one.

We have sprinkled calls of three *breakpoint handlers* throughout the mutually recursive clique of 52 functions implementing the ACL2 rewriter. These functions do nothing until break-rewrite is turned on with (brr t) or within the scope of with-brr-data. Logically the breakpoint handlers are no-ops that return nil. But when (brr t) has been done, a handler may enter a brr wormhole to save or erase data and to interact with the user.

- Near-miss-brkpt1 is called when the rewriter considers a rule but finds that the rule's left-hand side fails to match the current target. The brr wormhole is entered. The first thing that happens inside the brr wormhole when invoked by this handler is to determine whether the rule being considered by the rewriter is monitored and has near-miss criteria associated with it. These questions can only be answered from inside the wormhole since the rewriter has no information about monitored rules. If the answers are affirmative, the target is compared to the near-miss pattern of each specified near-miss criterion. If any near-miss pattern matches the target, the function pushes a new status on the stack of statuses, prints an "open break banner" explaining each of the near-misses just detected, and prompts the user for input. When the user issues the command to proceed from the break, the wormhole and near-miss-brkpt1 are exited. The rewriter continues as it would had the handler never been called. It will, in fact, subsequently call brkpt2 discussed below on the very same rewrite call stack. That will allow brkpt2 to detect that it should print a "close break banner" and pop the brr-status stack.

[3] This is a change made recently; for the first 30 years of break-rewrite's implementation the status was factored differently and when moved by wormhole from its persistent location to its ephemeral location was scattered over four different state globals. Now the status object is assigned to a single global.

- `Brkpt1` is called when the rewriter considers a rule and finds that the rule's left-hand side matches the current target. The `brr` wormhole is entered. If the rule is monitored and the break condition is satisfied, the function pushes a new status on the stack, prints an open break banner explaining the break, and prompts the user for input. When the user types an exit command the `brr-status` is updated to record which exit command was used. The `:eval` command means proceed to try to apply the rule and reenter this interactive break when the attempt is complete, `:go` means proceed, print the results of the attempt but do not interact further with the user, and `:ok` means proceed and do not even print the results. There are variants for controlling whether further breaks are allowed while attempting to apply the current rule. In all cases, these commands cause the wormhole and `brkpt1` to exit. The rewriter proceeds as though `brkpt1` had never been called, trying to relieve the hypotheses, test heuristic conditions, etc. It will eventually call `brkpt2` on the same call stack.
- `Brkpt2` is called when the rewriter is finished considering a rule. The `brr` wormhole is entered, with information passed in from the rewriter that includes the rewriter's call stack and data about what happened, e.g., whether the attempt succeeded or not, if not, why not, etc. If the rewriter's call stack is the same as the `brr-gstack` in the current `brr-status`, then we know this is the balancing "closing" phase of that open break. In this case, `brkpt2` either prompts the user for more input (if the opening break was exited with `:eval`) or just prints the appropriate close break banner, pops the `brr-status`, and exits the wormhole. In the case that `brkpt2` prompts the user for input then the closing banner, the stack pop, and exit happen when the user issues an exit command.

Thus each `near-miss-brkpt1` call that opened a break is balanced by a `brkpt2` call that closes it, and each `brkpt1` call that opened a break is balanced by a `brkpt2` call that closes it.[4] But there can be additional `near-miss-brkpt1`, `brkpt1` and `brkpt2` calls between a balanced pair that signals breaks, all from rewrite rule applications that are subsidiary to the one handled by that balanced pair, as we now illustrate.

Suppose we are in the state discussed in Section 3 where we had the first two rules noted below, but add the third rule.

```
(defthm p-rule  (implies (q x) (p (f x y))))
(defthm q-rule1 (implies (r x) (q x)))
(defthm q-rule2 (implies (s x) (q x)))
```

Monitor the first two rules as before, issue the `thm` command below, and type `:GO` to every break. The left column below shows all calls of `brkpt1` and `brkpt2` as obtained by tracing those two functions and just printing their names. We have further annotated those calls with the name, in brackets, of the lemma being considered by the rewriter when the breakpoint handler is called. Thus, "`> BRKPT1 {p-rule}`" means we enter `brkpt1` with `p-rule` being considered by the rewriter, and "`< BRKPT1 {p-rule}`" means we exit `brkpt1` with `p-rule` being considered. The right column shows the output of break-rewrite and the user's responses. We have indented the break output and deleted some blank lines.

```
ACL2 !>(thm (implies (r u) (p (f u v))))
1> BRKPT1 {p-rule}
                              (1 Breaking (:REWRITE P-RULE) on (P (F U V)):
                              1 ACL2 >:GO
<1 BRKPT1 {p-rule}
1> BRKPT1 {q-rule2}
```

[4] Care is taken to clean up the `brr-status` stack in the event of an error exit or interrupt.

```
<1 BRKPT1 {q-rule2}
1> BRKPT2 {q-rule2}
<1 BRKPT2 {q-rule2}
1> BRKPT1 {q-rule1}
                                    (2 Breaking (:REWRITE Q-RULE1) on (Q U):
                                    2 ACL2 >:GO
<1 BRKPT1 {q-rule1}
1> BRKPT2 {q-rule1}
                                    2 (:REWRITE Q-RULE1) produced 'T.
                                    2)
<1 BRKPT2 {q-rule1}
1> BRKPT2 {p-rule}
                              1 (:REWRITE P-RULE) produced 'T.
                              1)
<1 BRKPT2 {p-rule}
Q.E.D.
```

It might be surprising that each trace is at level 1: no call of `brkpt1` or `brkpt2` is within any other such call, even though the apparent calls of break-rewrite are nested. Break-rewrite is an illusion. No such function is defined in ACL2 (which is why we never write it in typewriter font).

The break-rewrite *depths* printed in the banners and prompts correspond to the `brr-previous--status` chain (via the brr stack depth). The break by `brkpt1` at depth 1, when the rewriter is considering `p-rule`, pushes a new status on the `brr-status` stack, prints a banner opening the break, reads the user's `:go`, and exits. The rewriter proceeds to try to relieve the hypothesis, `(q u)`, of `p-rule`. First it tries `q-rule2`. But when `brkpt1` is called on the unmonitored `q-rule2`, `brkpt1` just exits silently. The rewriter fails to relieve the hypothesis of `q-rule2` and calls `brkpt2` on `q-rule2`, which exits silently since the rewriter's call stack is not the `brr-gstack` of the status. Next the rewriter tries `q-rule1`, calling `brkpt1` on `q-rule1`, which is monitored. `Brkpt1` pushes another status on the `brr-status` stack making the depth 2. The `:go` at depth 2 exits that `brkpt1` and allows the rewriter to proceed to successfully establish the hypotheses of `q-rule1` and then call `brkpt2`. It detects that the rewriter's call stack is the `brr-gstack` of the status and that the balancing `brkpt1` at depth 2 proceeded with `:go`, so `brkpt2` prints the results, does not prompt for user input, prints the closing banner for depth 2, pops the `brr-status` stack, and exits. The same thing happens (at depth 1) when `brkpt2` is eventually called on `p-rule`.

7 Implementation aspects for `with-brr-data`

We have seen that `with-brr-data` supports *saving* of relevant data during a proof attempt, to be used by tools for *querying* the data. In this section we focus primarily on *saving* data, concluding with a few words about *querying* data.

Our approach to *saving* data is based on the observation that calls of `brkpt1` and `brkpt2` are in exactly the places we want to consider: before and after each matched rule is considered. As noted earlier, `with-brr-data` collects no data for near-miss breaks; thus, here we consider only `brkpt2` calls that balance `brkpt1` calls rather than balancing `near-miss-brkpt1` calls. (Technically, we restrict to those `brkpt2` calls for which the `failure-reason` argument is not the symbol, `near-miss`.) Note that unlike break-rewrite, there is no need for `brkpt2` to check the rewriter's call stack to check for balancing with a `brkpt1` call, since there is data collection for every `brkpt1` call.

Thus, `with-brr-data` piggybacks on `brr` in the sense that we modified those two breakpoint handlers to collect data when appropriate, to be queried later. But recall the focus on the source of a term in

a *checkpoint*. Therefore we consider only top-level rewrite rule applications, that is, we ignore rewrites that take place during backchaining. The functions in the ACL2 rewriter take an `ancestors` argument that is non-`nil` precisely during backchaining, so we restrict data collection to when `ancestors` is `nil`.

But how can we store global data when the rewriter does not modify the ACL2 state? As for break-rewrite, we use a wormhole state. We initially did this by modifying the `brr` wormhole status, but that resulted in very slow execution because, unlike typical uses of break-rewrite, `with-brr-data` saves data at *every* `brkpt1` and `brkpt2` call for which `ancestors` is `nil`. We solved this problem by making separate wormhole-eval calls to save data for `with-brr-data` into a different wormhole state, named `brr-data`.

That said, we still prefer to avoid calling even `wormhole-eval` when no data is to be stored. Here is the relevant code in `brpkt1`, with the `wormhole-eval` call abbreviated; the code in `brpkt2` is the same except that it uses `brkpt2-brr-data-entry` instead of `brkpt1-brr-data-entry`.

```
(and (eq gstackp :brr-data)
     (brkpt1-brr-data-entry ancestors gstack rcnst state)
     (wormhole-eval 'brr-data ...))
```

The first test is true in the scope of `with-brr-data`, as we'll discuss later. The second test is true when `ancestors` is `nil` (but the next section discusses how that can be changed). Only when those two conditions are met do we call `wormhole-eval` to store appropriate data in the `brr-data` wormhole.

The `wormhole-eval` call invokes functions to update the `brr-data` wormhole state: `update-brr--data-1` in `brkpt1` and `update-brr-data-2` in `brkpt2`. More precisely, the wormhole state consists of a list of `brr-data` records, which we now describe, and these two functions update that list.

A `brr-data` record contains fields `pre` and `post` that are `brr-data-1` and `brr-data-2` records, respectively; see below. `Pre` and `post` contain information from balanced `brkpt1` and `brkpt2` calls. A `brr-data` record also has a `completed` field, which is a list of `brr-data` records representing subsidiary rewrite rule applications (as further described below).

```
(defrec brr-data
  (pre post . completed)
  nil)

(defrec brr-data-1
  (((lemma . target) . (unify-subst . type-alist))
   .
   ((pot-list . ancestors) . (rcnst initial-ttree . gstack)))
  nil)

(defrec brr-data-2
  ((failure-reason unify-subst . brr-result)
   .
   (rcnst final-ttree . gstack))
  nil)
```

The `completed` field of a `brr-data` record B is a list of `brr-data` records created for balanced pairs of `brkpt1`/`brkpt2` calls that took place between `pre` and `post` fields, hence for the rewrite rule applications subsidiary to the one represented by B. Consider for example a rewrite rule r_1, `(equal (f1 x) (f2 x))`, and a rewrite rule r_2, `(equal (f2 x) (f3 x))`. So the application of rule r_1 to `(f1 a)` would generate an application of r_2 to `(f2 x)` with `x` bound to `a`, so that `(f3 a)` is the result of applying r_2 and hence of applying r_1 as well. This process would be recorded in a `brr-data` record whose `pre` field would have a `target` of `(f1 a)` and whose `post` field would have a `brr-result` field of `(f3`

a). Its completed field would have a single brr-data record representing the subsidiary application of rule r_2, hence with pre and post fields whose target and brr-result fields are (f2 a) and (f3 a), respectively.

The brr-data-1 and brr-data-2 structures include considerable information that isn't used by the built-in query utilities. However, with-brr-data doesn't cause much slowdown, and the extra data, made readily available by the formal parameters of brkpt1 and brkpt2, may be useful for user-defined attachments as discussed in the next section.

With-brr-data sets things up as follows to create brr-data records, as required for queries like cw-gstack-for-subterm.

1. Evaluate (clear-brr-data-lst) to remove previously saved data.
2. Assign state global gstackp to have value :brr-data. Note that in the test (eq gstackp :brr-data) above, the variable gstackp is the value of that state global.
3. Evaluate the argument of with-brr-data, to invoke the prover.
4. Set state global brr-data-lst based on the stored data, by calling (brr-data-lst state).

The last step is interesting in a couple of ways. First, note that with state global gstackp set to :brr-data, the prover populates the brr-data data wormhole state with a list of brr-data records, one for each rewrite rule application that is not subsidiary to any other (generally from rewriting a literal of a clause). The list is constructed in reverse order: as the proof proceeds, new records are pushed onto the front of that list. What's more, each completed field of each brr-data record is similarly in reverse order. So the last step above puts everything into the right order before storing the brr-data wormhole data into the state global, brr-data-lst. It may seem odd logically to obtain data from the brr-data wormhole outside that wormhole. The function get-persistent-whs provides the logical explanation by obtaining such data by reading the acl2-oracle field of the ACL2 state, which changes the state — though raw Lisp code for get-persistent-whs gets the result from the persistent wormhole status of the brr-data wormhole.

We conclude this section by commenting briefly on the implementation of the query utilities. These traverse the state global described above, brr-data-lst, searching for a brr-data record that represents a rewrite rule application introducing the specified subterm or term (or instance, in the :free case). The source code definitions of cw-gstack-for-subterm, cw-gstack-for-term, as well as their iterative ('*') versions, are all reasonably straightforward. Also see the documentation for with-brr-data for discussion of its keyword arguments and a few more implementation-level details.

8 Changing the behavior of with-brr-data

The preceding section mentions functions update-brr-data-1 and update-brr-data-2, which are invoked in brkpt1 and brkpt2, respectively, to update the list of brr-data records held in the brr-data wormhole state. These two functions are actually stubs that have respective attachments update-brr-data-1-builtin and update-brr-data-2-builtin, which implement the steps enumerated in the preceding section in a reasonably straightforward way.

In fact, with-brr-data was originally designed and implemented for collecting failed attempts at backchaining, rather than for collecting appropriate top-level rewrites as is done now. That original functionality is available by changing those attachments after including the community book, kestrel/utilities/brr-data-failures.lisp, and then issuing a single command. Below is that command

and its single-step macroexpansion. It shows how the argument provided to the command, in this case `failures`, provides a suffix for the attached function names.

```
ACL2 !>:trans1 (set-brr-data-attachments failures) ; whitespace edited below
 (WITH-OUTPUT :OFF :ALL
     (PROGN (DEFATTACH (UPDATE-BRR-DATA-1 UPDATE-BRR-DATA-1-FAILURES)
                       :SYSTEM-OK T)
            (DEFATTACH (UPDATE-BRR-DATA-2 UPDATE-BRR-DATA-2-FAILURES)
                       :SYSTEM-OK T)
            (DEFATTACH (BRKPT1-BRR-DATA-ENTRY BRKPT1-BRR-DATA-ENTRY-FAILURES)
                       :SYSTEM-OK T)
            (DEFATTACH (BRKPT2-BRR-DATA-ENTRY BRKPT2-BRR-DATA-ENTRY-FAILURES)
                       :SYSTEM-OK T)))
ACL2 !>
```

The community book `kestrel/utilities/brr-data-all.lisp` is similar except that it arranges to collect data for all rewrites, not just for failed backchaining. Just as the suffix "`failures`" was used above, the suffix "`all`" is used for this "`-all`" book. After including it, one would evaluate `(set-brr-data-attachments all)` to get the desired behavior via attachments.

Other behaviors can be implemented similarly. To take advantage of any of these, however, one might want to write suitable query utilities, perhaps modeled on the implementation of the existing query utilities such as `cw-gstack-for-subterm`.

9 Conclusion

Both `with-brr-data` and break-rewrite can be very useful tools in proof debugging. We have shown how to use them and we have given a glimpse of implementation issues and solutions.

Acknowledgments. This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. We also thank ForrestHunt, Inc. for supporting research reported herein, as well as the reviewers for helpful feedback.

References

[1] R. S. Boyer & J S. Moore (1988): *A Computational Logic Handbook*. Academic Press, New York.

[2] R. S. Boyer & J S. Moore (1997): *A Computational Logic Handbook, Second Edition*. Academic Press, New York.

[3] M. Kaufmann, J S. Moore & The ACL2 Community (2021): *The Combined ACL2+Books User's Manual*. `http://acl2.org/manual/index.html`.

[4] Matt Kaufmann & J Strother Moore (2020): *Iteration in ACL2*. *Electronic Proceedings in Theoretical Computer Science* 327, p. 16–31, doi:`10.4204/eptcs.327.2`.

Using Counterexample Generation and Theory Exploration to Suggest Missing Hypotheses

Ruben Gamboa
University of Wyoming
& Kestrel Institute*
Laramie, Wyoming
ruben@uwyo.edu

Panagiotis Manolios
Northeastern University
Boston, Massachusetts
p.manolios@northeastern.edu

Eric Smith
Kestrel Institute*
Palo Alto, California
eric.smith@kestrel.edu

Kyle Thompson
University of California San Diego
& Kestrel Institute*
San Diego, California
r7thompson@ucsd.edu

Newcomers to ACL2 are sometimes surprised that ACL2 rejects formulas that they believe should be theorems, such as `(reverse (reverse x)) = x`. Experienced ACL2 users will recognize that the theorem only holds for intended values of `x`, and given ACL2's total logic, there are many counterexamples for which this formula is simply not true. Counterexample generation (cgen) is a technique that helps by giving the user a number of counterexamples (and also witnesses) to the formula, e.g., letting the user know that the intended theorem is false when `x` is equal to 10. In this paper we describe a tool called DrLA that goes further by suggesting additional hypotheses that will make the theorem true. In this case, for example, DrLA may suggest that `x` needs to be either a `true-list` or a `string`. The suggestions are discovered using the ideas of theory exploration and subsumption from automated theorem proving.

1 Introduction

Over the past year, the Kestrel PEARLS team has been working to implement ideas to use machine learning tools to improve the experience of users in the construction and repair of proofs. Most of these ideas revolve around The Method [1]. In particular, we built an "advice" tool that can read an ACL2 checkpoint from a failed proof attempt, and suggest a number of routes the user may take to resolve the issue. The advice is created from a variety of models trained using machine learning techniques, as well as some heuristics that would be familiar to ACL2 users. The models are trained by taking data from the ACL2 Community Books, deliberating breaking the theorems in those books, and submitting the broken theorem to ACL2. The model is trained to recognize the checkpoint that ACL2 discovers when trying to prove the broken theorem, and then suggest the fix that corresponds to the way the theorem was originally broken. For example, one way to break a theorem is to remove an `include-book`, so the proposed fix is to include that particular library book.

There are, of course, various ways to break theorems, thus various different solutions that the advice tool may suggest. Besides removing library books, we may remove hints, or remove hypotheses from the theorem itself. Two lessons we learned while doing this are that (1) sometimes the advice tool could produce valuable advice using only hardcoded suggestions instead of full-blown machine learning (e.g.,

*This work was supported by a grant from the Defense Advanced Research Projects Agency (DARPA) Proof Engineering, Adaptation, Repair, and Learning for Software (PEARLS) Artificial Intelligence Exploration (AIE) Opportunity

A. Coglio, S. Swords (Eds.): 18th ACL2 Workshop (ACL2-2023)
EPTCS 393, 2023, pp. 82–93, doi:10.4204/EPTCS.393.8

"try enabling all definitions"), and (2) advising the user to add a hypothesis is inherently riskier than suggesting including a book or adding a hint. The second point is simply unavoidable, since adding a hypothesis *changes* the logical meaning of the intended theorem. For instance, suggesting the hypothesis `NIL` will result in a successful proof attempt of a useless theorem. But a less blatant problem would be suggesting the new hypothesis `x <= 0` in a theorem that already has the hypothesis `x >= 0`; the new theorem, which applies only to the case `x = 0` may be easier to prove, but it is also much less useful.

In light of this, and further considering the first point above, it is natural to ask whether there are simple strategies—i.e., not based on machine learning or artificial intelligence in general—that can be used effectively to suggest missing hypotheses. This paper introduces DrLA, a tool that does precisely this. Rather than use checkpoints from a failed proof attempt, DrLA infers missing hypotheses by using the counterexample generation engine (cgen) originally developed in ACL2s [4]. Cgen provides the user with both counterexamples and witnesses to the proposed theorem, and DrLA uses theory exploration techniques [3] to suggest the missing hypotheses. A key concept in theory exploration is to consider only terms that do not reduce to terms that have already been seen. In the context of generating hypotheses, this deals effectively with the problem of suggesting hypotheses that trivialize the original theorem.

The rest of the paper is organized as follows. Sect. 2 describes the necessary background from theory exploration to then introduce the key ideas behind hypothesis generation with DrLA. This is followed in Sect. 3 with a discussion of the effectiveness of DrLA (and similar tools). Then Sect. 4 discusses some details of the DrLA implementation. Finally, Sect. 5 provides some concluding remarks and suggests avenues for the future evolution of DrLA and other tools to ease proof development with ACL2.

2 Background and the Key Idea

Theory exploration is a technique for discovering likely properties of programs or lemmas of a mathematical theory. For example, once the function `append` over lists is defined, theory exploration may discover that append is, in fact, associative. This is done by combining two tools: a formula generator and a property checker (also known as a counterexample generator).

The formula generator creates formulas, i.e., possible theorems, from a given set of function and constant symbols. In the case of list functions, it may start with `consp`, `nil`, `cons`, `car`, `cdr`, `append`, and `equal`. Using this vocabulary, the theory exploration tool may create some familiar theorems such as

- `(equal (car (cons x1 x2)) x1)`
- `(equal (append (append x1 x2) x3) (append x1 (append x2 x3)))`

as well as reasonable-looking formulas that are not theorems, e.g.,

- `(equal (car (cons x1 x2)) x2)`
- `(equal (append x1 x2) (append x2 x1))`

and complete nonsense, such as

- `(consp (equal (car nil) (append x1 x2)))`
- `(car (cons (cdr x1) (equal x2 nil)))`

Theory exploration then considers each of these formulas in turn, and determines which of them are likely to be true. This is where the property checker comes in, by methodically searching for counterexamples to each formula. E.g., the first conjecture is `(equal (car (cons x1 x2)) x1)`, and it has the

variables `x1` and `x2`, so the property checker will consider thousands of random or strategically chosen values for them, such as `x1=3`, `x2='(1 2)` or `x1='(a . 16)`, `x2='bgs`. In all cases, the formula ends up being true, so theory exploration will suggest this formula as a *likely* lemma. Theorem proving can then be used to confirm that it is an *actual* lemma, and some theory exploration systems do this.

In the case of spurious theorems, theory exploration can often find an assignment that demonstrates the formula cannot be true. E.g., the binding `x1=3`, `x2='(1 2)` from above suffices to show that the formula `(equal (car (cons x1 x2)) x2)` cannot be true, so it would never be suggested as a likely lemma. The same is true of the nonsensical formulas, though care must be taken with respect to runtime errors, since these expressions may violate guards freely.

So the result of theory exploration is a list of theorems, or at least likely conjectures. The goal is to produce enough formulas that the tool can find a sufficient number of useful theorems. Obviously there is a delicate balance involving the formula generator. Ideally, it should generate as many formulas as possible, so that useful lemmas can be discovered. But the process of generating random formulas grows exponentially with their length, so limits are unavoidable, and efficient strategies are used to prune the space of candidate formulas so that barren areas of the search space are not explored. In practice, this means that the theorems discovered are usually small syntactically, e.g., limited in terms of depth.

For our purposes, we are interested in discovering not likely lemmas, but likely hypotheses that may be missing from a theorem. The overall strategy remains the same: A term generator will produce candidate hypotheses, and a property checker can determine if each possible hypothesis is likely to make the theorem provable. But a key idea is that we can leverage the work of the property checker since the hypotheses are always in the context of a surrounding formula, as opposed to theory exploration where the generated formulas are all at the top level. E.g., consider the motivating example `(equal (reverse (reverse x)) x)`. It is a reasonable heuristic to expect that any missing hypothesis will feature only the variable `x`, so we can generate values for `x` ahead of time and test all candidate hypotheses with the same set of bindings.

In fact, we can do a bit better than that. Cgen, the counterexample generator developed as part of ACL2s, is a sophisticated tool that will find both counterexamples and witnesses to an ACL2 formula. For our motivating example, cgen will identify the following counterexamples

- `((X '((T . 1) NIL . #\A)))`
- `((X '(-25 . 0)))`
- `((X '(53 . 252)))`

and the following witnesses

- `((X '((T T) (#\A 1))))`
- `((X NIL))`
- `((X '(-1)))`

Readers experienced with ACL2 will immediately recognize that all of the witnesses are true lists, whereas none of the counterexamples are—which immediately suggests `(true-listp x)` as the missing hypothesis.

DrLA proceeds in a similar manner. The basic idea is to find an expression that is false for all of the counterexamples and true for all witnesses[1]. This evokes the machine learning idea of finding a "hyperplane" that separates the positive and negative examples in a training data set. This is straightforward to do with a general property checker, as in theory exploration.

[1] We will see later that it is not necessary, or even desirable, for the property to hold for all witnesses, but this is a good first approximation.

The more interesting component is the term generator. What should be the language (i.e., function symbols) that determines the possible terms that lead to possible hypotheses? It is often the case that the missing hypothesis is a type hypothesis. ACL2 is an untyped language, but many functions are written with specific types in mind. This is certainly the case where `reverse` is concerned, since the obvious programmer intent is for `reverse` to work with lists, more specifically true-lists. Theorems about these functions usually require some typing hypothesis to make explicit the intended use of the function, and such hypotheses are easy to miss.

The tau system is an important component of ACL2 that helps the theorem prover benefit from the implicit notions of type assumed by programmers [2]. Tau is designed "to be a lightweight, fast, and helpful decision procedure for an elementary subset of the logic focused on monadic predicates and function signatures" [2]. So a reasonable language for the term generator is the set of types ("monadic predicates") in use by the tau system. This set begins with a hard-coded list of primitive predicates, including `consp`, `natp`, and so on. The set is enhanced when new definitions are encountered, e.g., `primep`, but only monadic predicates are considered. To account for the fact that sometimes the result of comparisons is a useful notion of type, tau hardcodes some common comparisons as primitive type predicates, e.g., `0 <= x`, `1 < x`, and "`x` is a non-NIL true-listp." DrLA adopts the primitive types recognized by the tau system, but unlike the tau system, it does not currently expand this dictionary as new functions are introduced. Rather, it uses the the predicates mentioned in the theorem and used in the definitions of functions present in the theorem. For example, when considering `(equal (reverse (reverse x)) x)`, it will add the function `reverse` to the list of types, as well as functions used in its definition such as `revappend`. This illustrates another departure from the tau system, in that `revappend` is a binary function, so it would be ignored by the tau system.

DrLA will take these selected symbols and generate terms by nesting syntactically valid function invocations. These terms are the expression trees that can be generated using these function symbols up to a maximum depth. The leaves of the terms correspond to either variable symbols, which must occur in the original theorem, or one of a list of predefined constant symbols, e.g., 0 or NIL. Thus, DrLA will explore terms including the following:

- `(posp x)`
- `(consp x)`
- `(reverse x)`
- `(revappend x 0)`
- `(equal (reverse x) x)`

DrLA will also consider boolean combinations of these terms, up to a maximum depth. For example, DrLA may consider the hypothesis `(or (posp x) (consp x))`. DrLA allows the user to control all of the depth parameters: how many levels of function definitions to explore for new function names, how deeply to nest boolean expressions, and how deeply to nest non-boolean terms such as `(reverse (reverse x))`.

We will have more to say about the implementation of DrLA in Sect. 4, but it is important to note now that DrLA attempts to avoid unnecessary computation when exploring the search space. For instance, DrLA is aware that the boolean predicates `and` and `or` are commutative, so it considers only one of the terms `(and P Q)` or `(and Q P)`. Moreover, it avoids nesting the primitive predicates, so that it will consider `(posp (reverse x))` but never `(posp (consp x))`.

As described thus far, DrLA will respond to the motivating example with an excess of possible hypotheses. Included in this list is the expected `(true-listp x)`. But DrLA also finds other possible suggestions, such as

- `(equal x 'nil)`
- `(equal (revappend x x) 'nil)`
- `(true-listp (revappend x x))`
- `(and (consp x) (true-listp x))`
- `(and (true-listp x) (equal (reverse x) 'nil))`

Actually, DrLA finds dozens of similar, unhelpful suggestions.

Thus, the final component of DrLA is a filter that reduces the number of suggestions in a manner reminiscent of subsumption. Specifically, suppose that DrLA has two suggestions P and Q such that $P \rightarrow Q$ but $Q \not\rightarrow P$. For example, P may be `(natp x)` while Q is `(integerp x)`. We say in this case that Q is more general (logically weaker) than P, and we prefer to suggest Q. With this heuristic, DrLA will suggest `(true-listp x)` but not `(equal x 'nil)` since `(true-listp x)` is the more general term. This heuristic eliminates many of the useless suggestions above, but not all. For instance, `(true-listp (revappend x x))` is logically equivalent to `(true-listp x)`, so neither is more general than the other. In these cases, a complexity heuristic comes into play—if P and Q are logically equivalent and Q is syntactically simpler than P, DrLA will suggest Q but not P. Our notion of syntactic complexity is simple and partial, so it is possible that DrLA will find two suggestions that are logically equivalent and just as simple syntactically. In these cases, DrLA will offer both suggestions, letting the user pick which one to use as the "better" hypothesis.

As mentioned previously, this notion of "more general" is similar to the notion of subsumption in resolution theorem proving. In the implementation of DrLA, it is used in much the same way. First, a suggestion is ignored if it is subsumed by a prior suggestion (c.f. forward subsumption.) Then, when a suggestion is added to the list of suggestions, prior suggestions subsumed by the new one are discarded (c.f. backward subsumption.)

We note briefly that these heuristics also have a useful side-effect. In Sect. 1, we warned about the dangers of suggesting vacuous hypotheses, such as `nil`. In fact, DrLA would never make such a suggestion, since such hypotheses would never be true of all the witnesses. But DrLA may suggest overly constrained hypotheses, such as `(equal x 'nil)`. However, the heuristics described will rule out that hypothesis in favor of the more general `(true-listp x)`.

With these heuristics, DrLA suggests a single hypothesis to the user, but it is *not* the expected `(true-listp x)`. Instead, DrLA suggests that the correct theorem is

```
(implies (or (stringp x)
             (true-listp x))
         (equal (reverse (reverse x)) x))
```

We believe that most users of ACL2 think of `reverse` as a function that operates on lists, but in fact the programmers of this function allowed for both lists and *strings*. The heuristics of DrLA select this hypothesis since it is more general than the expected `(true-listp x)`.

3 Assessment

In this section we describe our efforts to assess the performance of DrLA. The approach is to give DrLA versions of theorems from the Community Books, after removing one or more of the hypotheses in the theorem. This is the same idea used to assess the more general advice tool that we built as part of the PEARLS project, but with an important difference. The other suggested fixes address the proof itself,

e.g., by suggesting a hint. DrLA, on the other hand, suggests a modification to the theorem, e.g., a new hypothesis to add. DrLA is never invoked unless cgen demonstrates that the theorem is false, i.e., by finding some counterexamples.

What this means for assessment is that it is insufficient simply to check if the the suggestion results in a successful proof attempt. What is really important is to determine whether the suggestion matches the original intent of the ACL2 user, who presumably forgot to list one of more key hypotheses. Simply comparing the suggested hypothesis with the original one, i.e., the one that was removed earlier, is also insufficient, because there may be more than one way to express the necessary constraint.

So we chose to do much of the assessment manually, by running DrLA on examples from the Community Books and determining whether DrLA's output is effective. For example, the book `std/lists/append.lisp` is part of the lists library in ACL2, and it starts with the theorem

```
(implies (consp x)
         (< 0 (len x)))
```

Removing the hypothesis leaves just `(< 0 (len x))`, and when this is submitted to DrLA, it provides a single suggestion: `(consp x)`. Moreover, DrLA reports that after adding this hypothesis to the theorem, ACL2 is indeed able to find a proof. In this specific example, DrLA's performance is an unqualified success.

The next theorem in the file is

```
(implies (not (consp x))
         (equal (append x y)
                y))
```

When DrLA is prompted with the conclusion, it suggests `(atom x)` as the missing hypothesis. Note that `atom` is defined in ACL2 as not `consp`, so this is 100% consistent with the original theorem, although not identical. This is why we think a manual assessment process is necessary.

A more interesting theorem is

```
(iff (append x y)
     (or (consp x)
         (consp y)))
```

DrLA does not support `iff`, but we can use this by breaking the theorem up into two implications. When presented with `(append x y)`, DrLA accurately suggests `(or (consp x) (consp y))`, but it also makes a number of other suggestions which, while valid, are less useful. For instance, it suggests `(or (acl2-numberp y) (consp x))`. Notice that neither of these suggestions subsumes the other, and neither is syntactically simpler than the other, so DrLA suggests both[2] and lets the user decide which is the best alternative.

In the other direction, DrLA is completely ineffective. There are many witnesses and counterexamples for `(or (consp x) (consp y))`, but DrLA fails to find a single suggestion. The reason is that cgen finds many counterexamples to this formula, but it actually does not find any witnesses. In any case, the conclusion is a disjoint of simple types, so DrLA would be hard-pressed to find a *simpler* formula to suggest. It would certainly not even explore terms with the function symbol `append`.

The situation is somewhat better in the case of the following theorem:

```
(implies (and (not (index-of k x))
              (index-of k y))
         (equal (index-of k (append x y))
                (+ (len x) (index-of k y))))
```

[2] Actually, DrLA gives six total suggestions in this case.

When DrLA is presented with just the conclusion of this theorem, it discovers the missing hypotheses `(and (not (index-of k x)) (index-of k y))`. This reason this succeeds, is precisely that the key predicate, `index-of`, appears in the conclusion of the theorem, so DrLA knows to generate possible hypotheses that involve `index-of`.

In contrast, consider the following theorem, also from the standard lists library:

```
(implies (member k x)
         (equal (nth (index-of k x) x)
                k))
```

DrLA will consider terms that feature the primitive typing predicates, as well as any functions used in the theorem and the definition of functions that appear in the theorem, e.g., `nth` and `index-of`. However, `member` does not appear in the theorem or any of those definitions, so DrLA will fail to suggest the hypothesis of the theorem. These situations, where the missing hypothesis requires a predicate that does not otherwise appear in the theorem, are common.

To address this, DrLA allows the user to provide a list of additional predicate symbols that it should also consider. If `member` is provided in this way, DrLA may suggest `(member k x)` as a possible missing hypothesis. Unfortunately, DrLA's subsumption heuristic leads DrLA astray in this scenario. The problem is that any term that looks like `(or (member k x) P)` subsumes `(member k x)` regardless of what `P` is. Thus, DrLA suggests terms like the following instead of the correct `(member k x)`:

- `(or (member k x) (and (rationalp x) (< x 0)))`
- `(or (complex-rationalp x) (member k x))`

But in fact, these weaker hypotheses are not sufficient to prove the theorem; when `x` is 1/2, for example, `(nth (index-of k x) x)` is equal to `nil`, not necessarily equal to `k`. But the reason this shows up as a candidate missing hypotheses is that the candidates are chosen empirically by examining the counterexamples and witnesses—not by invoking the theorem prover directly. What DrLA lacks here is a large enough sample of witnesses and counterexamples, e.g., at least one counterexample where `x` is a rational or complex rational. So DrLA provides the user with 11 total suggestions, though it does inform the user that only two of them are sufficient to prove the original theorem, and those two end up being vacuous, i.e., with `x=nil`.

As these examples make clear, it is vital to the success of DrLA to start with a suitable set of function symbols so the forest of possible hypotheses is rich enough to contain useful candidates. Allowing the user to provide some candidates helps DrLA to succeed in some difficult cases, but it also feels inappropriate: If the user knows that `member` is a suitable hypothesis, surely she can simply provide the necessary hypothesis without using DrLA at all. This is an open issue that we will return to.

4 Implementation Notes

In this section, we describe some of the implementation decisions and tradeoffs that we encountered implementing DrLA. As previously mentioned, the first step is to invoke cgen using the function `prove/cgen` to generate a list of witnesses and counterexamples, and we ask cgen for 50 of each. If cgen cannot find any witnesses or any counterexamples, DrLA reports an error and does not attempt to find missing hypotheses.

Otherwise, DrLA proceeds by generating the expressions that may become candidate hypotheses. Although there is no such distinction in ACL2, DrLA considers Boolean patterns, predicates, and terms separately. Boolean patterns are partial expressions such as

- `(or NIL NIL)`
- `(and (or NIL NIL) NIL)`
- `(and (not NIL) NIL)`
- `NIL`

The `NIL`s are placeholders where an arbitrary predicate can be placed. These are just slots; there are no restrictions on the way predicates may appear, and in particular there is no requirement that the same predicate be used to replace multiple `NIL`s. Note: The significance of the last template, a single `NIL`, will become clear shortly.

DrLA then collects the primitive type predicates (as in the tau system), essential comparators (such as `equal` and `<<`), and any extra predicates suggested by the user (e.g., `member` above). Then it generates templates, similar to the Boolean patterns, using these names. For example, DrLA may generate terms such as

- `(equal NIL NIL)`
- `(integerp NIL)`
- `(member NIL NIL)`

Unlike the case with the Boolean expressions, DrLA does not nest these templates, since they are intended to be Booleans that operate on terms. (Recall that DrLA makes a distinction between Boolean patterns, predicates, and general terms.) The intent is to avoid considering unpromising expressions such as `(integerp (memberp X (equal Y Z)))`.

DrLA then combines these two template lists into a single list of templates, which may include such entries as

- `(equal NIL NIL)`
- `(or (equal NIL NIL) (integerp NIL))`

Note that each template is created by replacing one of the `NIL`s in the Boolean templates by one of the predicate templates (and now the reason for the single `NIL` Boolean template should be clear.) There is an exponential explosion here, so care is taken to use only tail-recursive functions to avoid running out of stack space. Using DrLA's default values, the total number of templates formed at this stage is 2,380—but that number can go up, e.g., if the user specifies a larger depth for boolean expressions.

Next, DrLA generates the term templates, which are similar to the above, but based on the functions that are present in the theorem or that appear in the definitions of those functions, up to a certain depth limit. For example, for the theorem

```
(equal (index-of k (append x y))
       (+ (len x) (index-of k y)))
```

DrLA will construct terms from the functions `APPEND`, `INDEX-OF`, `+`, `CDR`, `LEN`, and `BINARY-+`. Thus, it will generate the following 13 templates:

- `(+ NIL NIL)`
- `(append NIL NIL)`
- `(binary-+ NIL NIL)`
- `(car NIL)`
- `(cdr NIL)`

- `(eqlablep NIL)`
- `(index-of NIL NIL)`
- `(index-of-aux NIL NIL NIL)`
- `(index-of-aux-eq NIL NIL NIL)`
- `(index-of-aux-eql NIL NIL NIL)`
- `(len NIL)`
- `(return-last NIL NIL NIL)`
- `NIL`

In essence, the next step is to combine these templates with each of the 2,380 previously generated templates. However, that would lead to a veritable explosion of resulting templates, since many of the earlier templates have more than one placeholder `NIL`, e.g., the template `(or (equal NIL NIL) (integerp NIL))` which has three `NIL`s, so it would result in $13^3 = 2197$ templates by itself. In a language with lazy evaluation, this would not be a major problem, but with ACL2 we were concerned about space limitations if we tried to generate all the templates.

Instead, DrLA proceeds as follows. For each template, it determines how many placeholders it has, e.g., 3. Then it computes the total number of possible results, which in this case would be $13^3 = 2197$. Then it uses a counter to go through all of the possibilities, and generates the corresponding template just-in-time. E.g., a few of the templates generated may be[3]

0. [000] `(or (equal (+ NIL NIL) (+ NIL NIL)) (integerp (+ NIL NIL)))`
1. [001] `(or (equal (+ NIL NIL) (+ NIL NIL)) (integerp (append NIL NIL)))`
12. [00c] `(or (equal (+ NIL NIL) (+ NIL NIL)) (integerp NIL))`
13. [010] `(or (equal (+ NIL NIL) (append NIL NIL)) (integerp (+ NIL NIL)))`

The important point is that DrLA does not generate all of these at once. Rather, it generates the first, fully processes it (as described below), and then loops to generate the next template.

A further optimization is performed at this stage, which helps to manage the combinatorial explosion. Consider a term like `(and (integerp NIL) (integerp NIL))`. We call these terms redundant, and the key observation is that such a term was formed from a Boolean template, `(and NIL NIL)` in this case, and a predicate, `(integerp NIL)` in this case. But since `NIL` is a Boolean template in its own right, then `(integerp NIL)` will also be generated as a possible hypothesis—thus, we can safely ignore the more complicated but logically equivalent `(and (integerp NIL) (integerp NIL))`. It's not just duplicates that cause this, e.g., consider `(and (integerp X) (acl2-numberp X))` where one subterm implies the other. DrLA recognizes these cases and will ignore any template with a redundant subterm, since it knows a simpler, equivalent term will also be considered.

These are still templates, however, not full ACL2 terms. The next step is to consider the leaves that may be placed in each individual template. These are the free variables appearing in the original theorem, and possibly[4] a handful of constants such as 0, 1, `t`, `nil`. In our running example, the variables are `K`, `X`, and `Y`. The final step in term generation is to use these variables and constants to fill in the placeholders in the templates generated above. Again, this is done lazily by filling in each template with each possible n-tuple of values before moving to the next. The resulting terms will include the following (and many, many more):

[3] In the list, the numbers refer to the counter in decimal and [base13].)

[4] By default, DrLA does not use the built-in binary predicates `equal` and `<<`, in which case it also does not consider built-in constants. This reflects an attempt to manage the combinatorial explosion of term generation.

- `(or (equal (+ X X) (+ X X)) (integerp (+ X X)))`
- `(or (equal (+ K Y) (+ Y X)) (integerp (+ X K)))`

DrLA then processes each term to determine which should be candidate hypotheses. Recall that DrLA started by gathering a set of counterexamples and witnesses from cgen. Each counterexample or witness consists of a list of bindings for the free variables in the original theorem. E.g., a counterexample may look like

```
((K 3) (X '(1 2 3)) (Y '(10 20 30)))
```

and a witness may look like

```
((K 10) (X '(1 2 3)) (Y '(10 20 30)))
```

The important point is that witnesses and counterexamples fully bind the terms under consideration, so DrLA can use ACL2's executable interpreter to determine the value of each term under that assignment. We use `trans-eval` for this purpose, although there are other hooks into the ACL2 interpreter. What remains, then, is to check if the term evaluates to false for all of the counterexamples. If that's the case, then

```
(implies TERM
         BROKEN-THEOREM)
```

may be an actual theorem, so the term may be a good candidate hypothesis. At the very least, it eliminates all the (known) counterexamples.

Of course, we want the term to be satisfiable so that adding it as a hypothesis does not make the final theorem vacuously true. This is where the witnesses come in. Ideally, the proposed hypothesis will be true for all the witnesses. Such a hypothesis neatly separates the witnesses from the counterexamples (always true for the former and false for the latter).

However, we found that this is not always desirable. Our intuition is that when a user submits a theorem to ACL2, she has an expectation of the types of the variables under consideration, or some constraint on the possible values of them. The reason the proof attempt failed, however, is that this expectation was not explicit in the hypothesis, so the theorem as stated is in fact false. The counterexamples clearly attest to that, but the witnesses do not necessarily do the same. For example, consider the (false) theorem

```
(equal (<= (* k x) (* k y))
       (<= x y))
```

Here, the user forgot to specify that `K` is a non-negative integer. However, cgen may find witnesses such as `k=-1`, `x=0`, and `y=0`. In fact, it may find witnesses such as `k=-1`, `x=NIL`, and `y=NIL`. We call these witnesses, which are necessarily outside of the user's intent, "false witnesses." Insisting that any proposed hypothesis fully separates the counterexamples from the witnesses is too strict a requirement in the presence of false witnesses, so DrLA considers a hypothesis to be a good candidate if it is false for all the counterexamples, and true for at least one witness.

As mentioned previously, DrLA performs a subsumption check to eliminate as many candidate hypotheses as possible. The subsumption check is not just syntactic (as in traditional subsumption) but semantic, since it involves logical entailment. It would be possible to use the theorem prover itself to determine this, but we chose not to for two reasons. First, the subsumption checks happen often, as each potential hypothesis is compared to all other hypotheses. Making these many calls to the prover may simply prove impractical. Second, the theorem prover is not complete, so it may fail to prove that one hypothesis does in fact subsume another. Instead, we chose to use the witnesses and counterexamples

gathered from cgen to test logical entailment using the executable interpreter in ACL2. The same tradeoff calculation led to using the interpreter to determine when a template is redundant, as mentioned previously. Additional experiments are necessary to determine the right tradeoffs, so these design decisions may change in the future.

The last step is reporting the final list of candidate hypotheses to the user. Here, DrLA invokes the theorem prover on each "fixed" theorem, to see which of the suggested hypotheses in fact succeed in fixing the proofs. DrLA displays all suggestions to the user, not just the ones that resulted in successful proof attempts, since the real goal isn't a "Q.E.D.", but making explicit the hidden assumption that the user had forgotten. It may well be that one of the failing suggestions is actually closer to what the user intended, and seeing that suggestions will inspire the user to correct the theorem.

5 Conclusion and Further Improvements

This paper introduced DrLA, a tool that can help ACL2 users fix broken theorems by suggesting a forgotten hypothesis. This is done by borrowing and repurposing ideas from theory exploration and machine learning.

Our experience suggests that DrLA is a promising tool, though more work is needed before it can reach its potential. Some problems are technological. For example, since DrLA relies on counterexample generation, it can be only be used on formulas that are fully executable, i.e., no encapsulates. This problem may be addressed effectively using defattach.

Other problems are about efficiency. There is a combinatorial explosion at the heart of DrLA. Computers are getting faster, but combinatorics can't just be brushed aside. There are certainly more algorithmic tricks we can use to reduce the number of templates generated. The reader may have noticed, for example, that the templates described in Sect. 4 had entries for both `+` and `binary-+`. Detecting such redundancies could result in significant speedups.

Other improvements may come from revisiting DrLA's heuristics. This is particularly true of the selection of function names to be used in the term templates. DrLA looks for function symbols in the theorem itself, and in the definitions of functions used in the theorem. Some of this is necessary, or DrLA would only ever be able to suggest simple typing hypotheses. But as the examples showed, DrLA wastes a lot of time considering possible terms that are highly improbable, e.g., `eqlablep` or `index-of-aux-eql`.

On the other hand, DrLA sometimes fails to find any suitable hypotheses simply because it does not know to use certain functions. One possible way of addressing this is to follow the strategy of the tau system, which is to consider all unary predicates. (It is worth noting that unary predicates are especially effective at battling the combinatorial problem, since adding a unary predicate does not change the number of available slots.) However, this too could result in an excess of templates, since unary predicates are very common, and DrLA can't use the greedy optimizations that are so effective in the tau system. One possibility is to use an explicit system, as in the `defdata` types of ACL2. That is something we plan to explore in the near future. Another possible solution can be found by using machine learning, which is part of the broader context in which DrLA was developed. In particular, machine learning could be used to explore the Community Books for clusters of predicates in hypotheses that are associated with other predicates in conclusions of theorems. For example, theorems about binary trees may often use hypotheses about balanced trees. Even if "balanced" is not mentioned in the conclusion, DrLA could use such information to automatically consider it in such cases.

Finally, another place where DrLA could be improved is in the interaction with the counterexample

generator. Cgen makes extensive use of the known datatypes in the way that it searches for counterexamples. For instance, if the formula contains a variable `X` that is known to be a binary tree, cgen will try to find witnesses and counterexamples where `X` is bound to a binary tree. But this information is made available to cgen through the related `defdata` framework, which not all ACL2 users currently use. Addressing this is a major challenge, but one that is worthwhile. A better way of finding witnesses and counterexamples will immediately upgrade DrLA.

DrLA is an ongoing project, and we hope to continue improving it in the near future by following the roadmap described in this section.

References

[1] ACL2 Documentation (1996): *The Method*. `https://www.cs.utexas.edu/users/moore/acl2/v8-5/combined-manual/?topic=ACL2____THE-METHOD`. Accessed: 2023-07-17.

[2] ACL2 Documentation (2013): *Introduction to the Tau System*. `https://www.cs.utexas.edu/users/moore/acl2/v8-5/combined-manual/?topic=ACL2____INTRODUCTION-TO-THE-TAU-SYSTEM`. Accessed: 2023-07-17.

[3] Moa Johansson & Nicholas Smallbone (2021): *Conjectures, Tests and Proofs: An Overview of Theory Exploration*. *Electronic Proceedings in Theoretical Computer Science* 341, pp. 1–16, doi:10.4204/eptcs.341.1. Available at `https://doi.org/10.4204%2Feptcs.341.1`.

[4] Panagiotis Manolios (2013): *Counterexample Generation Meets Interactive Theorem Proving: Current Results and Future Opportunities*. pp. 18–18, doi:10.1007/978-3-642-39634-2_4.

Formal Verification of Zero-Knowledge Circuits

Alessandro Coglio Eric McCarthy Eric W. Smith
Kestrel Institute https://kestrel.edu
Aleo Systems Inc. https://aleo.org

Zero-knowledge circuits are sets of equality constraints over arithmetic expressions interpreted in a prime field; they are used to encode computations in cryptographic zero-knowledge proofs. We make the following contributions to the problem of ensuring that a circuit correctly encodes a computation: a formal framework for circuit correctness; an ACL2 library for prime fields; an ACL2 model of the existing R1CS (Rank-1 Constraint Systems) formalism to represent circuits, along with ACL2 and Axe tools to verify circuits of this form; a novel PFCS (Prime Field Constraint Systems) formalism to represent hierarchically structured circuits, along with an ACL2 model of it and ACL2 tools to verify circuits of this form in a compositional and scalable way; verification of circuits, ranging from simple to complex; and discovery of bugs and optimizations in existing zero-knowledge systems.

1 Introduction

In cryptography, a *zero-knowledge proof* is a method by which a *prover* can convince a *verifier* that they know a secret x that satisfies a computable predicate P, without revealing x and without involving third parties [22, 10]. Spurred by recent advances that have greatly improved their efficiency [9, 23, 14, 7, 13], zero-knowledge proofs are finding increasingly wide application, particularly in the blockchain world [20, 32, 8, 19, 12, 1], holding promise to rebalance privacy on the Internet [37, 18].

While most of the technical details of zero-knowledge proofs are irrelevant to this paper, the one crucial fact is that the predicate P must be expressed as a *zero-knowledge circuit*, which can be defined[1] as a set of equality constraints over integer variables where the only operations are addition and multiplication modulo a large prime number. This is a low-level representation P_{L} of P, at odds with the need for P to be clearly understood by both prover and verifier, who we presume would understand a higher-level representation P_{H} of P, e.g. expressed in a conventional programming language. Unless P_{L} and P_{H} denote the same P, the zero-knowledge proof may not quite prove what is expected.

This leads to the mathematically well-defined problem of formally proving that a zero-knowledge circuit correctly represents a higher-level description. Note the difference between formal proofs, which provide logic-based unconditional evidence of mathematical assertions, and zero-knowledge proofs, which provide cryptography-based statistically overwhelming evidence of computational assertions. Besides formal proofs about zero-knowledge proofs, which is the topic of this paper, one could imagine doing zero-knowledge proofs of formal proofs (i.e. prove a theorem without revealing the proof, which may have interesting applications), but we have not explored that yet. Given the above characterization of the problem in terms of zero-knowledge circuits, the zero-knowledge proof aspect is largely irrelevant here; the unqualified 'proof' and similar words in the rest of this paper have the familiar meaning.

This paper describes our endeavors, in the course of various projects, to tackle the zero-knowledge circuit verification problem, using ACL2 [26] and tools built on it. Our contributions are:

(a) A general formal framework for zero-knowledge circuit correctness, i.e. that $P_{\mathrm{L}} \Longleftrightarrow P_{\mathrm{H}}$.

(b) A library of rules to reason about *prime fields*—the arithmetic basis for zero-knowledge circuits.

[1] There seems to be no universal definition of zero-knowledge circuits and of some related notions in the literature.

A. Coglio, S. Swords (Eds.): 18th ACL2 Workshop (ACL2-2023)
EPTCS 393, 2023, pp. 94–112, doi:10.4204/EPTCS.393.9

(c) A formal model of *Rank-1 Constraint Systems* (*R1CS*), an existing formalism commonly used to represent zero-knowledge circuits.
(d) Rules and tools to verify R1CS circuits, including a new specialized version of Axe [38, axe].
(e) A formal model of *Prime Field Constraint Systems* (*PFCS*), a novel formalism developed by us that generalizes R1CS with richer forms of constraints and with hierarchical structure.
(f) Rules and tools to verify PFCS circuits, in a compositional and scalable way.
(g) Verification of zero-knowledge circuits, ranging from simple to complex, in R1CS and PFCS form.
(h) Discovery of two bugs and several optimizations in a zero-knowledge circuit construction library.

Section 2 provides the necessary background on zero-knowledge circuits. Section 3 describes contribution (a). Section 4 describes contribution (b). Section 5 describes contributions (c), (d), (g), and (h). Section 6 describes contributions (e), (f), and (g). Related work is discussed in Section 7. Future work is outlined in Section 8. Some conclusions are drawn in Section 9.

2 Background

A *prime field* is a set $\mathbb{F}_p = \{0, \ldots, p-1\}$, consisting of the natural numbers below p, where p is a prime number. The *arithmetic operations* on $\mathbb{F}_p$ are:

addition: $x \oplus_p y = (x + y) \bmod p$
subtraction: $x \ominus_p y = (x - y) \bmod p$
multiplication: $x \otimes_p y = (x \times y) \bmod p$
division: $x \oslash_p y = z$, where $x = y \otimes_p z$, if $y \neq 0$

That is, all the operations are modular versions of the ones on the integers, except that the division of x by y yields the unique z (which always exists) that yields x when multiplied by y, provided that $y \neq 0$. We may denote the prime field arithmetic operations with the same symbols as the integer arithmetic operations, i.e. $+$, $-$, $\times$, $/$. We may also omit the multiplication symbol altogether, e.g. $(x+1)(y-1)$ may stand for $(x+1) \times (y-1)$, which in turn may stand for $(x \oplus_p 1) \otimes_p (y \ominus_p 1)$. We may also just write $\mathbb{F}$, leaving p implicit. Context should always disambiguate these commonly used abbreviations.

A *zero-knowledge circuit* is a set of *constraints* that are equalities between *expressions* built out of *variables*, *constants*, *additions*, and *multiplications*, all interpreted in $\mathbb{F}$. By designating certain variables as *inputs* and *outputs*, the constraints can represent a computation of outputs from inputs. Zero-knowledge circuits generalize *arithmetic circuits*, which are like boolean circuits, except that wires carry integers instead of booleans, and gates perform arithmetic operations instead of boolean ones.

For reasons that depend on the details of zero-knowledge proofs, such constraints must be written in specific forms [15]. A popular formalism is *Rank-1 Constraint Systems* (*R1CS*), whose constraints have the form $(a_0 + a_1 x_1 + \cdots + a_n x_n)\,(b_0 + b_1 y_1 + \cdots + b_m y_m) = (c_0 + c_1 z_1 + \cdots + c_l z_l)$, where $n, m, l \geq 0$, each a_i, b_j, c_k is a *coefficient* in $\mathbb{F}$, and each x_i, y_j, z_k is a *variable* ranging over $\mathbb{F}$. That is, an R1CS constraint is an equality between the product of two polynomials and a polynomial, each polynomial having zero or more variables with exponent 1, i.e. a *linear combination*. An R1CS circuit is a set of these constraints. Literature definitions of R1CS are usually in terms of vectors and matrices; the definition just given here is more like an abstract syntax of R1CS.

For example, if w is *boolean*, i.e. either 1 or 0, the circuit in the left part of Figure 1 represents the computation in the right part, which sets z to x or y based on whether w is 1 or 0. If $w = 0$, the left side of the constraint is 0 and thus $z = y$; if

$$(w)\ \ (x-y)\ =\ (z-y) \qquad z := \begin{cases} x & \text{if } w = 1 \\ y & \text{if } w = 0 \end{cases}$$

Figure 1: An 'if-then-else' conditional.

$w = 1$, the $-y$ cancels and thus $z = x$.

As another example, the circuit in the left part of Figure 2 represents the computation in the right part, which sets w to 1 or 0 based on whether $u = v$ or not. If $u = v$, the first constraint makes $w = 1$, and the second constraint is satisfied because $0 \times 1 = 0$. If $u \neq v$, the second constraint makes $w = 0$, and the first constraint is satisfied by $s = 1/(u-v)$.

$$\begin{array}{rcl} (u-v)\ (s) & = & (1-w) \\ (u-v)\ (w) & = & (0) \end{array} \qquad w := \begin{cases} 1 & \text{if } u = v \\ 0 & \text{if } u \neq v \end{cases}$$

Figure 2: An equality test.

These and other examples can be found in the literature, e.g. [30] and [24, Appendix A]. Circuits vary in size and complexity. Even the ones in Figure 1 and Figure 2 require a little thought to understand.

Larger circuits are built from smaller ones by joining their constraints and sharing some variables. For instance, combining Figure 1 and Figure 2 yields a circuit that represents the computation that sets z to x or y based on whether $u = v$ or not; the variable w is shared, with Figure 2 guaranteeing that it is boolean as assumed in Figure 1. In this kind of hierarchical construction, a *gadget* is a circuit with a well-defined purpose, usable as a component of larger gadgets, and possibly made of smaller gadgets. The zero-knowledge circuit P_{L} in Section 1 is a top-level gadget; it represents the computation, described in some high-level way P_{H}, of the predicate P on the secret input x.

While all the variables in the gadget in Figure 1 are involved in the represented computation, the variable s in the gadget in Figure 2 is not. It is an *internal* variable, while the other ones are *external* variables; the latter are divided into *input* and *output* variables according to the represented computation. When the two gadgets are combined as just described, the shared external variable w becomes internal to the combined gadget. The distinction between external and internal variables, and between input and output variables, is not captured in the R1CS formalism, but it is arguably implicit in the notion of gadget. In general, internal variables cannot be avoided in gadgets; attempts to eliminate them often result in subtly non-equivalent constraints that fail to adequately represent the intended computation.

Although direct support is limited to $\mathbb{F}$ as a data type and (field) addition and multiplication as operations, R1CS circuits are at least as expressive as boolean circuits: if x and y are boolean variables (like w earlier), the constraint $(x)(y) = (1-z)$ represents a 'nand' gate with output z (and similarly simple constraints represent other logical gates); and higher-level data types can be always encoded as bits. But more efficient representations (fewer variable and constraints) are often possible.

For example, two unsigned n-bit integers, encoded as the bits $x_0, \ldots, x_{n-1}$ and $y_0, \ldots, y_{n-1}$ in little endian order, can be added via the gadget in Figure 3. The first $n+1$ constraints force $z_0, \ldots, z_n$ to be boolean. The last constraint forces them to be the bits of the sum, in little endian order, where z_n is the carry. This assumes that the prime p has at least $n+2$ bits, so that the field operations do not wrap around p; a typical p has about 250 bits, sufficient for fairly large integers.

$$\begin{array}{rcl} (z_0)\ (1-z_0) & = & (0) \\ & \vdots & \\ (z_n)\ (1-z_n) & = & (0) \\ (\sum_{i=0}^{n} 2^i z_i)\ (1) & = & (\sum_{i=0}^{n-1} 2^i x_i + \sum_{i=0}^{n-1} 2^i y_i) \end{array}$$

Figure 3: An unsigned n-bit integer addition.

R1CS circuits are normally constructed programmatically using libraries [3, 4, 6, 27, 28] that provide facilities to build gadgets hierarchically. These libraries are invoked directly, by programs written to build specific circuits, or indirectly, by compilers of higher-level languages to R1CS [2, 5, 25, 31, 11, 29]. As these libraries are invoked, they generate growing sequences of the R1CS constraints that form the gadgets.[2] Separate instances of the same gadget have different variables, which are typically generated

[2]The final sequence consists of the constraints for the predicate P in Section 1, and is part of the zero-knowledge proof.

via monotonically increasing indices. The gadgets' hierarchical structure, reflected in both the static organization and the dynamic execution of the libraries, is lost in the generated flat sequence of constraints; this is not an issue for zero-knowledge proofs, but it can be for formal proofs, as elaborated later.

3 Formal Framework

An R1CS circuit, along with an ordering of the r variables that occur in it, determines a relation $R \subseteq \mathbb{F}^r$ consisting of the r-tuples that satisfy all the constraints, when assigned element-wise to the variables. If additionally the variables are partitioned into q external ones and $r-q$ internal ones (in the sense of Section 2), and ordered so that the former precede the latter, a relation $\widetilde{R} \subseteq \mathbb{F}^q$ is also determined, defined as $\widetilde{R} = \{\langle \phi_1, \ldots, \phi_q \rangle \mid \exists \langle \phi'_1, \ldots, \phi'_{r-q} \rangle . R(\phi_1, \ldots, \phi_q, \phi'_1, \ldots, \phi'_{r-q})\}$, i.e. consisting of the q-tuples that satisfy all the constraints, when assigned element-wise to the external variables, for some $(r-q)$-tuples assigned element-wise to the internal variables. The tuples are *solutions* of the constraints. The informal notion of *gadget* described in Section 2 can be more precisely defined as an R1CS circuit accompanied by an ordering and designation of its variables as just described; $\widetilde{R}$ is the semantics of the gadget. For example, for the gadget in Figure 2, $R = \{\langle u, v, w, s \rangle \mid (u-v)\, s = 1 - w \wedge (u-v)\, w = 0\}$ and $\widetilde{R} = \{\langle u, v, w \rangle \mid \exists s . R(u, v, w, s)\}$.

Given this semantic characterization, it is natural to use a relation $S \subseteq \mathbb{F}^q$ as *specification* of the gadget, and to express *correctness* of the gadget as $\widetilde{R} = S$. The specification S may be defined in whichever high-level way that is convenient (more on this later), but in any case it denotes the set of q-tuples that must be the solutions of the gadget. Correctness consists of *soundness* $\widetilde{R} \subseteq S$ (i.e. every solution of the gadget satisfies the specification) and *completeness* $S \subseteq \widetilde{R}$ (i.e. everything satisfying the specification is a solution of the gadget). To prove soundness and completeness, the definition of $\widetilde{R}$ must be expanded, to expose the constraints that define R. To prove soundness, the existential quantification over the antecedent can be turned into a universal quantification over the implication, leading to the quantifier-free formula $R(\phi_1, \ldots, \phi_q, \phi'_1, \ldots, \phi'_{r-q}) \Longrightarrow S(\phi_1, \ldots, \phi_q)$. To prove completeness, no such move is possible: the formula $S(\phi_1, \ldots, \phi_q) \Longrightarrow \exists \langle \phi'_1, \ldots, \phi'_{r-q} \rangle . R(\phi_1, \ldots, \phi_q, \phi'_1, \ldots, \phi'_{r-q})$ demands dealing with the existential quantification explicitly, typically by exhibiting witnesses for the internal variables $\phi'_1, \ldots, \phi'_{r-q}$.

When a gadget represents a computation (as in Section 2), the specification S must specify the computation. For this, a *computation* is modeled as a function $f : I_1 \times \cdots \times I_n \to (O_1 \times \cdots \times O_m) \cup \{\mathcal{E}\}$ from $n \geq 0$ inputs to $m \geq 0$ outputs or to a distinct error $\mathcal{E}$. The case $n = 0$ is uninteresting but unproblematic. The case $m = 0$ models assertion-like computations, e.g. for each gadget $(z_i)(1 - z_i) = (0)$ that checks whether z_i is a bit, used in Figure 3: the computation represented by the gadget returns the empty tuple of outputs $\langle\,\rangle$ if z_i is boolean, or $\mathcal{E}$ otherwise. The function f always models a deterministic computation, which is appropriate for zero-knowledge applications.[3] The function f only captures the computation's input/output behavior, not other aspects of its execution; this is consistent with the fact that R1CS constraints only express relations among variables. For example, for the gadget in Figure 2, $I_1 = I_2 = O_1 = \mathbb{F}$, $f(u, v) = 1$ if $u = v$, $f(u, v) = 0$ if $u \neq v$, and thus $f(u, v) \neq \mathcal{E}$ always.

To represent f in R1CS form, its inputs and outputs must be represented as field elements, via injective encoding functions $e_i^{\mathrm{I}} : I_i \to \mathbb{F}^{n_i}$, where e_i^{I} maps each input $x_i \in I_i$ to some number n_i of field elements, and $e_j^{\mathrm{O}} : O_j \to \mathbb{F}^{m_j}$, where e_j^{O} maps each output $y_j \in O_j$ to some number m_j of field elements. This leads to a computation on encoded inputs and outputs $\widehat{f} : \mathbb{F}^N \to \mathbb{F}^M \cup \{\mathcal{E}\}$, with $N = \sum_i n_i$ and $M = \sum_j m_j$, defined as follows: (1) if $f(x_1, \ldots, x_n) = \langle y_1, \ldots, y_m \rangle$ then $\widehat{f}(e_1^{\mathrm{I}}(x_1), \ldots, e_n^{\mathrm{I}}(x_n)) = \langle e_1^{\mathrm{O}}(y_1), \ldots, e_m^{\mathrm{O}}(y_m) \rangle$;

[3] While zero-knowledge proofs themselves involve non-deterministic computations, normally they (probabilistically) prove facts about deterministic computations.

(2) if $f(x_1,\ldots,x_n) = \mathcal{E}$ then $\widehat{f}(e_1^{\mathrm{I}}(x_1),\ldots,e_n^{\mathrm{I}}(x_n)) = \mathcal{E}$; and (3) $\widehat{f}$ returns $\mathcal{E}$ outside the range of the input encodings. For example, for the gadget in Figure 2, $e_1^{\mathrm{I}} = e_2^{\mathrm{I}} = e_1^{\mathrm{O}} = id$ (identity) and $\widehat{f} = f$.

The computation $\widehat{f}$ is represented by a gadget with $q = N+M$ external variables for the inputs and outputs. The specification of $\widehat{f}$ is $S(x_1,\ldots,x_N,y_1,\ldots,y_M) = [\widehat{f}(x_1,\ldots,x_N) = \langle y_1,\ldots,y_M\rangle]$. Thus, soundness means that every solution of the gadget corresponds to a non-erroneous instance of the computation, and completeness means that every non-erroneous instance of the computation corresponds to a solution of the gadget. Soundness alone is not sufficient for correctly representing a computation: a gadget without solutions is trivially sound; completeness ensures that there is a solution for every input for which the computation is not erroneous. For example, for the gadget in Figure 2, $S = \{\langle u,v,w\rangle \mid f(u,v) = w\}$.

For a top-level gadget P_{L} that represents P in a zero-knowledge proof (see Section 1 and Section 2), whose formal semantics is a relation $\widetilde{R}_P$ as above, the specification S_P is derived from the high-level description P_{H} of P. If P_{H} is a program in a higher-level language [2, 5, 25, 31, 11, 29], a function f_P that denotes the execution of the program is formally defined, based on a formalization of the language, and a specification relation S_P is derived from it as above. If P_{H} is a description of a fixed application-specific computation (e.g. Zcash shielded transactions [19]), f_P is defined by formalizing that description, and S_P is derived from it as above. For sub-gadgets of P_{L}, specifications may be written in any formal form that is convenient; these specifications play a role in the formal verification of P_{L} (see Section 5.5 and Section 8), but they are not directly exposed to the zero-knowledge prover and verifier mentioned in Section 1. The understandability of P_{H} by prover and verifier, mentioned in Section 1, as with all complex technologies, boils down to trusting authoritative high-level informal descriptions for users from the general public, analyzing the aforementioned programs for users who are also software developers, and examining the formalizations and theorems for users who are also formal verification specialists.

4 Prime Fields

Starting with the recognizer of prime numbers `primep` from [39, `[books]/projects/numbers`], the *prime fields library* [38, `prime-fields`] introduces a recognizer `fep` of field elements, and functions `add`, `sub`, `mul`, `div`, `neg`, `inv`, `pow`, and `minus1` for field operations. These are all parameterized over a prime `p`, e.g. `(fep x p)` checks if `x` is in $\mathbb{F}_{\mathtt{p}}$, and `(add x y p)` returns $\mathtt{x} \oplus_{\mathtt{p}} \mathtt{y}$ (see Section 2).

The recognizer and operations are executable. The multiplicative inverse `inv` is calculated via `pow`, according to the known equation $1 \oslash_p x = x^{p-2} \bmod p$; we prove that this definition indeed yields the multiplicative inverse. The definition of `pow` is an `mbe` whose `:logic` is recursively repeated multiplication and whose `:exec` is fast modular exponentiation `mod-expt-fast` from [38, `arithmetic-3`].

The library provides basic theorems, such as all the standard field axioms (e.g. commutativity of addition); these theorems often suffice for relatively simple reasoning. The library also provides collections of rules that realize certain normalization strategies, useful for more elaborate reasoning.

5 Rank-1 Constraint Systems

5.1 Model

Based on the prime fields library described in Section 4, the *R1CS library* [38, `r1cs`] provides an ACL2 model of R1CS. It follows the nomenclature of literature definitions of R1CS, which are in terms of vectors and matrices. The model consists of a *dense* formulation, where linear combinations have monomials for all the involved variables (many with zero coefficients), and a *sparse* formulation, where linear

combinations may omit monomials with zero coefficients. The dense formulation is of intellectual interest but impractical for verification; the rest of this paper focuses on the sparse formulation.

The model formalizes a *pseudo-variable* as either a *variable* (an ACL2 symbol) or the number 1. A linear combination is formalized as a (sparse) *vector*, i.e. an ACL2 list of pairs (ACL2 lists of length 2) where each pair consists of a *coefficient* (a field element) and a pseudo-variable; the notion of pseudo-variable provides uniformity between monomials of degrees 1 and 0 in linear combinations. A *constraint* is formalized as an aggregate [38, `defaggregate`] with components `a`, `b`, and `c` that are the three linear combinations; this corresponds to the equality $(\mathtt{a})\,(\mathtt{b}) = (\mathtt{c})$, referring to Section 2. Finally, a (*rank-1 constraint*) *system* is formalized as an aggregate consisting of a prime, a list of variables, and a list of constraints. The model also defines well-formedness conditions on this aggregate and its sub-structures, e.g. that all the variables in the constraints are also in the list of variables of the R1CS aggregate. This aggregate and its sub-structures form the model's formal *syntax* of R1CS.

The *semantics* of R1CS is formalized in terms of satisfaction of constraints by *valuations*, which are ACL2 alists from variables to field elements, i.e. assignments of field elements to variables. Given a valuation, a linear combination evaluates to a field element, in the obvious way; the model defines this evaluation in terms of *dot product* of vectors, as in the literature. Given a valuation, a constraint evaluates to a boolean, in the obvious way. A valuation *satisfies* a system iff it makes all its constraints true.

Besides basic theorems about the syntax and semantics sketched above, the R1CS library also includes rules for reasoning about R1CS, for both ACL2 and Axe (see below).

5.2 Extraction

To verify gadgets using the R1CS model described above, the gadgets must be represented in the syntactic form defined by the model. The gadget construction libraries mentioned in Section 2 produce R1CS constraints that are not in that form, and sometimes they do not provide facilities to export them in any form. Thus, the approach to extract gadgets for verification is case by case.

In a Kestrel project funded by the Ethereum Foundation, we worked on the verification of Ethereum's Semaphore circuit [38, `semaphore`]. Since Semaphore was written in the high-level language Circom [25], whose compiler had a facility to export the R1CS constraints in JSON format, we developed an ACL2 converter from that format [39, `[books]/kestrel/ethereum/semaphore/json-to-r1cs`], and we used that along with our ACL2 JSON parser [39, `[books]/kestrel/json-parser`]. Taking advantage of the modularity of the Circom code, we extracted not only the complete circuit gadget, but also several sub-gadgets.

In a Kestrel project funded by the Tezos Foundation, we worked on the verification of Zcash's Jubjub elliptic curve operation circuits [38, `zcash`]. Since these circuits were generated programmatically in Rust, we instrumented that Rust code, with help from the Zcash team, to export R1CS constraints directly as s-expressions in the R1CS model's format. We extracted the top-level gadgets and several sub-gadgets, by invoking the library at different points.

At Aleo, we are working on the verification of the snarkVM gadgets [17]. With our Aleo colleagues, we have instrumented snarkVM's Rust code to export R1CS constraints in JSON format (different from Circom's). We have developed an ACL2 converter from that format to the model's format, which we are using along with the ACL2 JSON parser. We are extracting gadgets at varying levels of granularity.

In the current situation, in all the above cases, the gadget extraction is trusted: an error in our instrumentation of the gadget construction libraries, or in our conversion to ACL2, may cause us to unwittingly verify a different gadget from the real one. However, the top-level gadget P_{L} (discussed in Section 1, Section 2, and Section 3) is part of the zero-knowledge proof, which has a well-defined (protocol-dependent)

format, and is generated by tools like snarkVM [3] that use or include gadget construction libraries, That format can be formalized in ACL2, and the formal verification can be applied directly to (the P_L gadget in) the zero-knowledge proof. In this eventual situation, the extraction of sub-gadgets of P_L via instrumentation and conversion will merely provide building blocks for the top-level formal proof of P_L (especially in the compositional approach in Section 5.5), but will no longer be trusted.

5.3 Verification in ACL2

Regardless of the exact approach, the result of the above extraction is an ACL2 constant, say `*gadget*`, whose value is an R1CS aggregate of the form described in Section 5.1. The model confers semantics to this aggregate, amounting to the relation R in Section 3. More precisely, the model provides a predicate over an assignment of field elements to variables: `(r1cs-holdsp *gadget* asg)` means that the assignment `asg` satisfies all the constraints in `*gadget*`, given the prime that is part of the `*gadget*` aggregate (which is left implicit in R). This can be turned into a finitary relation over the field elements assigned to the variables, like R, by specializing `r1cs-holdsp` with an assignment to the gadget's specific variables, e.g. if the variables in `*gadget*` are `'x0`, `'x1`, ..., the relation R is formalized as

```
(defun gadget (x0 x1 ...)
  (r1cs-holdsp *gadget* (list (cons 'x0 x0) (cons 'x1 x1) ...))
```

where each `'xi` is a variable and each `xi` is a field element.

To state and prove correctness, a specification is written in ACL2, amounting to S in Section 3:

```
(defun spec (x0 x1 ...) ...)  ; this can be defined in any form
```

If the gadget has no internal variables, correctness is stated as

```
(defthm gadget-correctness
  (implies (and ...   ; boilerplate hypotheses
                ...)  ; preconditions (if applicable)
           (equal (gadget x0 x1 ...) (spec x0 x1 ...))))  ; R = S
```

where the boilerplate hypotheses say that `x0`, `x1`, ... are field elements, and where examples of preconditions are that some `xi` is boolean or that some `xi` is non-zero. If the gadget has internal variables, `spec` has fewer parameters, but soundness can be stated and proved similarly, using `implies` in place of `equal`. Completeness is less straightforward; it is discussed, in a more general context, in Section 5.5.

The proofs are carried out by first enabling certain functions of the R1CS semantics, so that the (evaluated) constraints *deeply embedded* in ACL2 are rewritten to ACL2 terms involving prime field operations, i.e. constraints *shallowly embedded* in ACL2. Then the core of the proof is handled via other hints and lemmas, of varying complexity, that depend on the details of the constraints and specification.

After verifying, in the manner just described, the correctness of a number of Semaphore sub-gadgets for elliptic curve operations and data multiplexing [39, `[books]/kestrel/ethereum/semaphore`], two related issues became apparent. One issue was that the numbers of variables and constraints and the resulting ACL2 terms grew quickly as we moved from simpler to more complex gadgets, making the proofs harder and less efficient. Another issue was that because each gadget was extracted in isolation, with its own specific variable names generated by the gadget construction libraries (typically via monotonically increasing indices), it was not easy to use proofs of sub-gadgets in proofs of super-gadgets: the same sub-gadget could appear with different variable names in different super-gadgets, or in different instantiations within the same super-gadget, but the proof for the separate sub-gadget used different variable names than would be seen in any of these instantiations.

5.4 Verification in Axe

To combat the growth of terms mentioned in Section 5.3, we turned to the Axe toolkit [38, `axe`]. The *Axe Rewriter* is functionally similar to the ACL2 rewriter, but it represents terms as directed acyclic graphs (DAGs) instead of trees: these DAGs share sub-terms, affording the practical handling of very large terms, such as fully unrolled AES implementations.

We developed a specialization of the *Axe Lifter* for R1CS, which turns deeply embedded constraints into shallowly embedded ones, similarly to what is described in Section 5.3, and also performs some simplifications of the lifted constraints using the Axe Rewriter. This specialized lifter [38, `lift-r1cs`] generates an ACL2 constant whose value is a DAG representing the simplified lifted constraints.

We developed a specialization of the *Axe Prover* for R1CS, which, given a DAG from the lifter as above and a specification like `spec` in Section 5.3, attempts to prove soundness [38, `verify-r1cs`] or completeness (via a more general event macro to prove implications). This specialized prover uses rewriting and variable elimination via substitution, and it supports applying different sets of rewrite rules in sequence. Substitution is enabled by the fact that certain constraints essentially equate certain variables to expressions over other variables, though rewriting must often be performed first to make this explicit by solving the constraints. A constraint is a candidate for substitution if it equates a variable with some sub-DAG not involving that variable. Large R1CS proofs can involve hundreds or thousands of substitution steps, and we optimized Axe to apply many substitutions at once when possible. For each round of substitution, Axe substitutes a set of variables each of which is equated to a sub-DAG not involving any variables in the set. The set of equalities used in the round is then removed from the assumptions of the proof. Repeated substitution of intermediate variables can incrementally turn a large unstructured conjunction of constraints into a deeply nested operator tree (represented in DAG form), of the kind commonly verified by Axe. The ability to apply the rewriting tactic with different sets of rewrite rules supports the staging of inter-dependent proof steps, which depend on previous steps and enable subsequent steps. Suitable rewrite rules can recognize R1CS idioms and turn them into equivalent higher-level formulations that may facilitate the rest of the proof. Similarly, certain sub-gadgets may also be recognized and raised in abstraction using rewrite rules based on the correctness properties of such sub-gadgets; this partially addresses the second issue described in Section 5.3.

The Axe verification of the soundness of an R1CS gadget looks like

```
(lift-r1cs *gadget-dag*  ; name of the generated defconst
           '(x0 x1 ...)  ; variables of the gadget
           ...           ; constraints of the gadget
           ...           ; prime of the gadget
           ...)          ; options
(verify-r1cs *gadget-dag*       ; gadget (simplified and lifted, in DAG form)
             (spec x0 x1 ...)   ; specification
             :tactic ...        ; proof tactics, e.g. (:rep :rewrite :subst)
             ...)               ; other information and options
```

We used this approach to verify the soundness, and in some cases also the completeness, of a number of Semaphore and Zcash (see Section 5.2) sub-gadgets that perform fixed-size integer operations, elliptic curve operations, instances of the MiMC cipher, and parts of the BLAKE2s hash [39, `[books]/kestrel/ethereum/semaphore`] [39, `[books]/kestrel/zcash/gadgets`]; these range from relatively small and simple to relatively large and complex. We also verified the soundness of the large and complex BLAKE2s hash gadget generated by (an earlier version of) snarkVM [3]; this is currently not open-source, but it will be in the future.

While using Axe helps address the term growth problem, the sub-gadget proof re-use problem remained largely unsolved. The recognition and rewriting of sub-gadgets mentioned above, which worked for certain cases, in general may need to recover the sub-gadgets from a sea of constraints. Each sub-gadget may consist of multiple constraints, some of which may even have the same form across different sub-gadgets, requiring the exploration of multiple recovery paths. Furthermore, constraint optimizations, such as the ones performed by the Circom compiler and by snarkVM, which blend gadgets under certain conditions, may greatly complicate, or defeat altogether, the recovery of sub-gadgets. Solving these problems is not necessarily impossible, but it is challenging; as a data point, the aforementioned soundness verification of snarkVM's BLAKE2s took several person-days to develop and takes several machine-hours to run.

5.5 Compositional Verification

As mentioned in Section 2, the hierarchical structure in the gadget construction libraries gets flattened away in the generated R1CS constraints. Thus, as discussed in Sections 5.3 and 5.4, the gadgets extracted from the libraries are verified as wholes, with limited ability to discern their hierarchical structure and leverage proofs of their sub-gadgets, resulting in difficult and slow proofs.

More scalability can be achieved via *compositional verification*, where the proof of a gadget uses the proofs of its sub-gadgets and is used in the proofs of its super-gadgets. This could be accomplished by extending the gadget construction libraries to generate such compositional proofs along with the gadgets, but doing so is impractical due to the libraries' complexity and ownership. A viable approach is to (1) replicate the gadget constructions in the theorem prover, (2) verify correctness properties of the constructions, and (3) validate the replicated gadget constructions by checking that the constructed gadgets are the same as the ones extracted from the libraries. We propound the term *detached proof-generating extension* for this kind of solution.

The gadget constructions are formalized by ACL2 functions that take variable names as inputs and return lists of R1CS constraints as outputs. The constraints are built either directly or by calling functions that build sub-gadgets, concatenating all the resulting constraints together. These functions return lists of constraints, which are readily composable by concatenation; they do not return R1CS aggregates (see Section 5.1), which are not readily composable. The gadget hierarchy corresponds to the function hierarchy. The parameterization over the variable names is critical, because separate instances of the same gadget have different variables, as mentioned in Sections 2 and 5.3.

To *validate* that these constructions are consistent with the libraries, we extract *sample gadgets* from the libraries as in Section 5.2, and we formulate ACL2 ground theorems saying that the extracted R1CS constraints are identical to the ones built by the ACL2 functions when passed suitable variable names as arguments. Currently this validation process amounts to testing our constructions against the libraries. Eventually, this validation will be performed every time the libraries are run to generate a zero-knowledge proof, as explained in Section 8.

The correctness of the ACL2 gadget constructions is proved for generic variable names and generic prime p (sometimes under restricting hypotheses). The proof opens the function definition and uses the theorems for any called functions, whose definitions are unopened; if the function builds some constraints directly, certain semantic functions of the R1CS model are also opened, lifting those constraints to equalities and prime field operations. Given this proof setup, the correctness of the gadget (family) built by the function is proved by reasoning over the specifications of the sub-gadgets (not the sub-gadgets' constraints) and/or the constraints of the gadget; the details depend on the gadget, and may involve hints and lemmas of varying complexity.

For example, a gadget to force a variable to be boolean as in Section 2 is constructed as

```
(defun boolean-assert-gadget (x)
  (list (make-r1cs-constraint :a (list (list 1 x))              ; (x)
                              :b (list (list 1 1) (list -1 x))  ; (1 - x)
                              :c nil)))                         ; (0)
```

where `x` is the variable name to use. Correctness (soundness and completeness) is expressed as

```
(defthm boolean-assert-gadget-correctness
  (implies ...  ; boilerplate hypotheses
           (equal (r1cs-constraints-holdp (boolean-assert-gadget x) asg p)  ; R
                  (bitp (lookup-equal x asg)))))                            ; S
```

where `asg` assigns field elements to variables, `lookup-equal` retrieves them, and `bitp` is the specification of this gadget; in the notation of Section 3, this theorem rewrites R ($= \widetilde{R}$ in this case) to S.

As another example, the gadget in Figure 2 is constructed as

```
(defun equality-test-gadget (u v w s)
  (append (list (make-r1cs-constraint ...))    ; (u - v) (s) = (1 - w)
          (list (make-r1cs-constraint ...))))  ; (u - v) (w) = (0)
```

Soundness is expressed as

```
(defthm equality-test-gadget-soundness
  (implies (and ...  ; boilerplate hypotheses
                (r1cs-constraints-holdp (equality-test-gadget u v w s) asg p))  ; R
           (equal (lookup-equal w asg)                                          ; S
                  (if (equal (lookup-equal u asg) (lookup-equal v asg)) 1 0))))
```

where the specification of this gadget is that the value of `w` is 1 or 0 based on whether the values of `u` and `v` are equal or not; in the notation of Section 3, this theorem derives S from R ($\neq \widetilde{R}$ in this case).

The gadget described in Section 2 as the combination of Figure 2 and Figure 1 is constructed as

```
(defun if-equal-then-else-gadget (u v x y z w s)
  (append (if-then-else-gadget w x y z)
          (equality-test-gadget u v w s)))
```

which calls the functions for the sub-gadgets (the definition of `if-then-else-gadget` is not shown).

To exemplify varying numbers of variables and constraints, the gadget in Figure 3 is constructed as

```
(defun addition-gadget (xs ys zs)
  ...  ; guard requires (len xs) = (len ys) = (len zs) - 1
  (append (boolean-assert-list-gadget zs)
          (list (make-r1cs-constraint
                 :a (append (pow2sum-vector xs) (pow2sum-vector ys))
                 :b (list (list 1 1))
                 :c (pow2sum-vector zs)))))
```

where `xs`, `ys`, and `zs` are lists of variables, `boolean-assert-list-gadget` constructs boolean constraints for all the variables in `zs`, and `pow2sum-vector` constructs a powers-of-two weighted sum. The parameterization covers not only the names of the variables, but also the number of bits n in Figure 3, which is the length of `xs` and `ys`. Correctness is expressed as

```
(defthm addition-gadget-correctness
  (implies (and ...                                    ; boilerplate hypotheses
                (< (1+ (len xs)) (integer-length p))   ; restriction on n
                (bit-listp (lookup-equal-list xs asg)) ; precondition
```

```
      (bit-listp (lookup-equal-list ys asg)))  ; precondition
 (equal (r1cs-constraints-holdp (addition-gadget xs ys zs) asg p)
        (and (bit-listp (lookup-equal-list zs asg))
             (equal (lebits=>nat (lookup-equal-list zs asg))
                    (+ (lebits=>nat (lookup-equal-list xs asg))
                       (lebits=>nat (lookup-equal-list ys asg)))))))
```

where `lebits=>nat` turns a list of bits into the integer they denote in little endian order, and where the restriction on n ensures that the modular weighted sums can be turned into non-modular sums. This is proved for every n, using a property of `pow2sum` proved by induction. While the proofs for the previously exemplified gadgets are straightforward, this gadget takes a little more work.

The details of the examples above, and of the other ones in Section 2, are in the supporting materials, in [39, `[books]/workshops/2023/coglio-mccarthy-smith`]. Other examples are in the R1CS library, in [39, `[books]kestrel/crypto/r1cs/sparse/gadgets`], where in particular the proofs in `range-check.lisp` were quite laborious.

We have employed this approach to verify compositionally a substantial portion of the snarkVM gadgets [17], specifically most of the ones for boolean, field, and integer operations. In the process, we have discovered two bugs in the gadgets, which have been fixed:[4] (i) the gadget to convert a field element into its bits failed to constrain the integer value of the bits to be below the prime, leading to indeterminacy (e.g. the field element 0 could be converted to not only all zero bits as expected, but also to the bits that form the prime, since $p \bmod p = 0$); and (ii) the gadget to calculate square root allowed both positive and negative roots (when the input is a non-zero square), leading to indeterminacy. We have also identified some possible optimizations, which have been or are being applied, saving a large number of constraints in some cases. Our ACL2 work on snarkVM is currently not open-source, but it will be in the future.

But even this approach eventually runs into a scalability issue, due to the internal variables of gadgets. The names of these variables are exposed as function parameters of not only the gadgets that directly use them to build constraints, but also any super-gadgets that contain (possibly many instances of) those sub-gadgets. As increasingly large gadgets are constructed, the function parameters for variable names keep growing, including all the internal variables at every level. Furthermore, while soundness theorems like `equality-test-gadget-soundness` above can ignore internal variables in the consequent of the implication, completeness theorems need to say something about the internal variables. In the notation of Section 3, a gadget correctness theorem $\widetilde{R} = S$ reduces to $R = S$ if there are no internal variables, which is a good rewrite rule, as in `boolean-assert-gadget-correctness` above. But internal variables cannot be existentially quantified in the gadget construction functions, because these functions must return the gadgets given all their variables. Instead, the specification S over the external variables must be extended to a specification S' over all variables, including the internal ones at every level. This exposure of internal variables violates modularity and impedes compositionality.

6 Prime Field Constraint Systems

The scaling issue discussed in Section 5.5 is addressed by *Prime Field Constraint Systems* (*PFCS*), a formalism introduced by the authors. PFCS generalizes R1CS in two ways: (1) constraints can be equalities between any expressions, built out of variables, constants, additions, and multiplications; and (2) constraints can be grouped into *named relations with parameters*, and these relations can be used as constraints with the parameters replaced by argument expressions (as in function calls).

[4] The Aleo blockchain mainnet had not been launched yet, so these bugs did not affect real applications and assets.

The first extension is useful to represent zero-knowledge circuit formalisms different from R1CS, but is not especially relevant to verifying R1CS gadgets. The second extension is important for verifying R1CS and other kinds of gadgets, because it explicitly captures their hierarchical structure. A PFCS relation formalizes a *gadget*; the relation's parameters are the gadget's external variables, while the other variables in the relation's defining body are the gadget's internal variables.[5] PFCS explicitly handles the existential quantification that takes R to $\widetilde{R}$: while R is the semantics of a PFCS relation's body, $\widetilde{R}$ is the semantics of the PFCS relation itself. The internal variables of a gadget are taken into consideration when proving the correctness $\widetilde{R} = S$ of a gadget, which involves R, but can be ignored when proving the correctness of super-gadgets that include that sub-gadget, whose semantics $\widetilde{R}$ can be rewritten to S in proofs for the super-gadgets; no extended specification S' (see end of Section 5.5) is needed.

Our development and use of PFCS is still somewhat preliminary. It is overviewed here, but it will be described in more detail in future publications. More information is in the PFCS library [38, `pfcs`].

6.1 Model

The *syntax* of PFCS is approximately described by the grammar in Figure 4, consistently with the informal description above. A relation R consists of a name N, a sequence of parameters N^*, and a defining body that is a sequence of constraints C^*. The abstract syntax is formalized via recursive types [38, `fty`]. The concrete syntax is formalized via an ABNF grammar [38, `abnf`] complemented by some (upcoming) restricting predicates.

names	N	::= ⟨letter then letters/digits/underscores⟩
integers	I	::= ... | -2 | -1 | 0 | +1 | +2 | ...
expressions	E	::= N | I | E+E | E*E
constraints	C	::= E=E | N(E*)
relations	R	::= N(N*){C*}

Figure 4: PFCS syntax.

The *semantics* of PFCS is approximately described by the inference rules in Figure 5, which inductively define when an assignment α (a finite map from variables to field elements) satisfies a constraint c in the context of a set of relations ρ, written $\alpha \vdash_\rho c$. The first rule says that α satisfies an equality constraint $e_1 = e_2$ when the evaluations $\alpha(e_1)$ and $\alpha(e_2)$ yield the same field element; α extends from variables to expressions in the obvious way. The second rule says that α satisfies a relation constraint $r(e_1 \cdots e_n)$ when ρ includes the relation $r(v_1 \cdots v_n)\{c_1 \cdots c_m\}$ and each constraint c_i in its body is satisfied by an assignment α' that extends the assignment of each evaluated argument expression $\alpha(e_j)$ to the corresponding parameter v_j of the relation; besides the parameters, α' must assign field elements to the other variables (if any) in the body of the relation, which are internal to the gadget. Since α' appears in the premises but not in the conclusion, it is existentially quantified; since the values of the relation's parameters are prescribed by the rule, the existential quantification reduces to the values assigned to the internal variables (if any), capturing exactly the existential quantification in $\widetilde{R}$. The prime p is left implicit in Figure 5.

$$\frac{\alpha(e_1) = \alpha(e_2)}{\alpha \vdash_\rho e_1 = e_2}$$

$$\frac{\begin{array}{c} r(v_1 \cdots v_n)\{c_1 \cdots c_m\} \in \rho \\ \alpha' \supseteq \{v_1 \mapsto \alpha(e_1), \ldots, v_n \mapsto \alpha(e_n)\} \\ \forall i \in \{1, \ldots, m\}.\ \alpha' \vdash_\rho c_i \end{array}}{\alpha \vdash_\rho r(e_1 \cdots e_n)}$$

Figure 5: PFCS semantics.

Since ACL2 disallows mutually recursive `defun` and `defun-sk`, the PFCS semantics is formalized, over the PFCS abstract syntax, via (1) proof trees for the inference rules in Figure 5 and (2) a proof

[5] PFCS does not distinguish between input and output external variables. This distinction only matters to the formulation of the specification S, which is still always a relation over the external variables, as is the semantics $\widetilde{R}$ of the gadget.

checker for those proof trees; that is, a mini-logic is formalized in ACL2. Since this definition is inconvenient for reasoning about gadgets, ACL2 rules are provided that capture the inference rules more directly, without proof trees and proof checker, as if `defun` and `defun-sk` were mutually recursive.

6.2 Verification

In the PFCS framework, gadget constructions are formalized by ACL2 functions that take no or few inputs and return (abstract syntax of) PFCS relations as outputs. Gadgets with fixed numbers of variables and constraints are built by ACL2 functions with no inputs. Gadgets with varying numbers of variables or constraints are built by ACL2 functions whose inputs are non-negative integers that specify those varying numbers. None of these ACL2 functions take variable names as inputs, because variables in PFCS relations are local to the relations and can be fixed for each gadget: the external variables, i.e. the parameters, can be replaced when the relations are called; and the internal variables are existentially quantified. These ACL2 functions do not call each other, unlike the ones that construct R1CS gadgets; the gadget hierarchy is captured directly in the PFCS relations.

Correctness is proved for generic prime p and (if applicable) for generic numbers of variables and constraints (sometimes under restricting hypotheses). The *deeply embedded* PFCS relations built by the ACL2 functions are lifted to *shallowly embedded* PFCS relations, which are ACL2 predicates over field elements, with parameters for the external variables and an existential quantification (via `defun-sk`) for the internal variables. These predicates are defined as conjunctions of (1) calls of other predicates, one per sub-gadget, and (2) equalities between terms involving prime field operations, one per equality constraint; the predicates' call graph corresponds to the gadget hierarchy. For gadgets with fixed numbers of variables and constraints, a deep-to-shallow *lifter* automatically generates the predicates, along with theorems connecting the deep and shallow formulations; for gadgets with varying numbers of variables and constraints, currently the predicate and theorem are manually generated, but a future extension of the lifter may automate these as well. Correctness of a shallowly embedded PFCS relation is proved by opening the predicate definition, using the called predicates' correctness theorems as rewrite rules, and using other hints and lemmas of varying complexity as needed. Correctness is extended to the deeply embedded PFCS relation via the lifting theorem, in a way that may be automated in the future.

For example, a PFCS version of `boolean-assert-gadget` in Section 5.5 is constructed as

```
(defun boolean-assert-gadget ()  ; deeply embedded PFCS relation
  (pfdef "boolean_assert"        ; name
         (list "x")              ; parameter
         (pf= (pf* (pfvar "x")                        ; (x)
                   (pf+ (pfconst 1) (pfmon -1 "x")))  ; (1 - x)
              (pfconst 0))))                          ; (0)
```

The lifter call `(lift (boolean-assert-gadget))` generates the predicate

```
(defun boolean-assert (x p)  ; shallowly embedded PFCS relation
  (and (equal (mul x (add (mod 1 p) (mul (mod -1 p) x p) p) p)
              (mod 0 p))))
```

and the lifting theorem

```
(defruled definition-satp-of-boolean-assert-to-shallow
  (implies ... ; boilerplate hypotheses
           (equal (definition-satp "boolean_assert" defs (list x) p)  ; deep
                  (boolean-assert x p))))                             ; shallow
```

where (definition-satp r ρ (list ϕ_1 $\cdots$ ϕ_n) p) formalizes $\{v_1 \mapsto \phi_1, \ldots, v_n \mapsto \phi_n\} \vdash_\rho r(v_1 \cdots v_n)$. The correctness of the predicate is expressed as

```
(defthm boolean-assert-correctness
  (implies ...  ; boilerplate hypotheses
           (equal (boolean-assert x p)  ; R (shallow)
                  (bitp x))))           ; S
```

which is extended to the gadget via the lifting theorem as

```
(defthm boolean-assert-gadget-correctness
  (implies ...  ; boilerplate hypotheses
           (equal (definition-satp "boolean_assert" defs (list x) p)  ; R (deep)
                  (bitp x))))                                         ; S
```

As another example, a PFCS version of `if-equal-then-else-gadget` in Section 5.5 is built as

```
(defun if-equal-then-else-gadget ()
  (pfdef "if_equal_then_else"
         (list "u" "v" "x" "y" "z")
         (pfcall "if_then_else" (pfvar "w") (pfvar "x") (pfvar "y") (pfvar "z"))
         (pfcall "equality_test" (pfvar "u") (pfvar "v") (pfvar "w"))))
```

The lifter generates the predicate

```
(defun-sk if-equal-then-else (u v x y z p)
  (exists (w)
          (and (fep w p)
               (and (if-then-else w x y z p)
                    (equality-test u v w p)))))
```

which existentially quantifies `w` and which calls the lifted predicates for its sub-gadgets (not shown here). Correctness is expressed as

```
(defthm if-equal-then-else-gadget-correctness
  (implies ...  ; boilerplate hypotheses
           (equal (definition-satp "if_equal_then_else" defs (list u v x y z) p)  ; R
                  (equal z (if (equal u v) x y)))))                               ; S
```

which rewrites R to S without involving the internal variable w.

To exemplify varying numbers of variables and constraints, a PFCS version of `boolean-assert-list-gadget` mentioned (but not shown) in Section 5.5 is constructed as

```
(defun boolean-assert-list-gadget (n)
  (pfdef (iname "boolean_assert_list" n)  ; "boolean_assert_list_<n>"
         (iname-list "x" n)               ; (list "x_0" "x_1" ...)
         (boolean-assert-list-gadget-aux  ; ((pfcall "boolean_assert" (pfvar "x_0"))
          (iname-list "x" n))))           ;  (pfcall "boolean_assert" (pfvar "x_1"))
                                          ;  ...)
(defun boolean-assert-list-gadget-aux (vars)
  (cond ((endp vars) nil)
        (t (cons (pfcall "boolean_assert" (pfvar (car vars)))
                 (boolean-assert-list-gadget-aux (cdr vars))))))
```

where `iname` constructs an indexed name, `iname-list` constructs a list of indexed names, and the auxiliary function constructs a list of PFCS relations calls for generic variable names (which is useful for induction), which the main function instantiates to specific variable names. Correctness is expressed as

```
(defthm boolean-assert-list-gadget-correctness
  (implies ...  ; boilerplate hypotheses
           (equal (definition-satp "boolean_assert_list" defs xs p)  ; R
                  (bit-listp xs))))                                  ; S
```

The details of the examples above, and of the other ones in Section 2, are in the supporting materials, in [39, `[books]/workshops/2023/coglio-mccarthy-smith`]. Other examples are in the PFCS library, in [39, `[books]kestrel/crypto/pfcs/examples.lisp`].

We are porting the verified snarkVM gadgets mentioned in Section 5.5 from R1CS form to PFCS form, which we will also use for the remaining snarkVM gadgets.

6.3 Validation

The PFCS gadget constructions in ACL2 are built in the same way as the R1CS gadget constructions in Section 5.5, namely by replicating what the gadget construction libraries do. For the ACL2 R1CS constructions, different choices of function call graph are possible, so long as they produce the same R1CS constraints as the libraries. For the ACL2 PFCS constructions, different choices of PFCS hierarchy are possible, so long as, when flattened, they produce the same R1CS constraints as the libraries.

We plan to develop a *flattener* of PFCS to R1CS, which will also generate theorems of correct flattening, i.e. that the flattened R1CS constraints are equivalent to the PFCS constraints. The flattener will inline all the relation constraints, resulting in a sequence of equalities, all of which have the R1CS form because our PFCS constructions use equality constraints of the R1CS form.

The PFCS gadget constructions in ACL2 will be validated against sample gadgets from the libraries in the same way as explained in Section 5.5 for the R1CS gadget constructions in ACL2, with the addition of the aforementioned PFCS flattener.

7 Related Work

The authors are not aware of any other work to formally verify zero-knowledge circuits using ACL2; the paper [17] describes our snarkVM verification work in more detail. There is work using other tools, discussed below.

The QED^2 tool [36] is a specialized verifier that combines a dedicated algorithm with an SMT solver to automatically establish whether the outputs of a zero-knowledge circuit are uniquely determined by the inputs, or are instead under-constrained; it may also fail to find an answer. Their approach is automated, but our work addresses a stronger property (correctness); the unique determination of outputs from inputs is implied by soundness, when the specification of a gadget is that the gadget represents a computation (see Section 3). Their approach works on individual circuits like the ones in Sections 5.3 and 5.4, not on parameterized circuit families like the ones in Sections 5.5 and 6.2.

The SMT solver for finite fields described in [34] has been used to verify automatically whether circuits produced by certain compilers are sound (with respect to the compilation source) and deterministic (i.e. the outputs are uniquely determined by the inputs, as in [36]). Since our circuit specifications prescribe computations, in a way that may be similar to the sources of circuit compilers, their soundness proofs are analogous to ours (with determinism implied by soundness, at least in our case, as noted above); but their work does not cover completeness proofs. Their approach works on individual circuits, not on parameterized circuit families (as also noted above for [36]). For example, in our work, an unsigned n-bit integer addition circuit family as in Figure 3 is verified once, quickly, for every possible n (see Section 5.5), and can be used to verify correct compilation via a syntactic check; in contrast, in their

work, instances of that family for different values of n are verified separately, taking increasing resources as n grows. Another advantage of verifying parameterized circuit families is that their definitions are essentially formal models of the circuit construction libraries and therefore help validate the libraries' design and implementation. The tradeoff between their and our approach is automation versus generality.

The Ecne tool [40] uses a dedicated algorithm to perform weak verification (their term to mean that the outputs are uniquely determined by the inputs) and witness verification (their term to mean that the outputs and the internal variables are uniquely determined by the inputs); their paper also discusses strong verification (i.e. correctness in our work), but only as future work. As already noted, the determinism of output variables is a consequence of soundness in our work. The determinism of internal variables is unnecessary for correctness, but it becomes a consequence of correctness if the latter is stated with respect to an extended specification S' that includes the internal variables (as in Section 5.5) and prescribes their computation from the inputs; Ecne's witness verification can be thus addressed with our techniques.

There is work on verifying the compilation of higher-level languages to zero-knowledge circuits [21, 35, 29]. While there is probably overlap with our work, and thus the opportunity for cross-fertilization, the purpose is a bit different: we verify the circuits constructed by existing libraries, which may be used as compilation targets, or for more general purposes such as programmatic construction of circuits; as noted above, our approach also helps validate the libraries.

As a final remark, the notion of existentially quantified circuits (EQCs) in [33] is related to the existential quantification of internal variables in PFCS.

8 Future Work

The main thread of future work is the continued verification of the snarkVM gadgets at Aleo, extending and improving the ACL2 PFCS library along the way. We plan to extend the PFCS lifter to work on parameterized gadgets, which requires a leap in sophistication in order to operate on the ACL2 functions that construct those gadgets rather than on the PFCS abstract syntax produced by the ACL2 functions. We also plan to build a proof-generating flattener of PFCS to R1CS form, to enable validation against samples extracted from snarkVM (see Section 6.3). To handle the gadget optimizations in snarkVM, we plan to develop proof-generating PFCS-to-PFCS transformations that correspond to those optimizations: these can be composed with the proofs for the vanilla (unoptimized) gadgets to obtain proofs of the optimized gadgets. The end goal is to verify all the snarkVM gadgets, including complex ones for cryptographic operations. These gadgets are being verified against specifications written in ACL2, which are not directly exposed to prover and verifier (cf. end of Section 3); they are building blocks for the next steps described below.

After reaching the above goal, the next verification target is the snarkVM compiler from Aleo instructions (the assembly-like language used to represent program code in the Aleo blockchain) to R1CS constraints, which uses the snarkVM gadget constructions to generate the constraints. The approach will be a detached proof-generating extension of the snarkVM compiler, built on the detached proof-generating extension of the snarkVM gadget constructions; every time the compiler is run, its generated constraints will be syntactically compared to the ones generated in ACL2, including flattening and optimizations as described above, to ensure that they are identical and thus that the proof applies to that exact compilation, as in a verifying compiler. The specification for the R1CS constraints generated by the snarkVM compiler is the source Aleo instructions program, which software developers can read and understand; the formal proof relies on a formalization of Aleo instructions that we are building in ACL2.

The same detached proof-generating extension approach will then be used for the compilation from

Leo [2] (a high-level programming language for the Aleo blockchain) to Aleo instructions, providing an end-to-end verifying compiler functionality from Leo to R1CS constraints via Aleo instructions. The specification for the R1CS constraints generated by the Leo and snarkVM compilers is the Leo source program, which software developers can read and understand; the formal proof relies on a formalization of Leo that we are building in ACL2 [16].

The roadmap delineated above is part of our overarching work to apply formal verification to ideally every aspect of the Aleo blockchain and ecosystem. The aforementioned formalizations of Aleo instructions and of Leo have more general value than their role in the formal verification of the compilation.

Alongside the PFCS-based compositional verification approach, it would also be interesting to continue exploring the Axe-based whole-gadget verification approach, in particular to improve the ability to recover sub-gadgets. There are tradeoffs between the two approaches: the first one keeps the proofs more manageable and efficient, but requires the formalization of the gadget constructions; the second one does not require that formalization, but needs to recover some of that structure during the proofs.

It would also be interesting to investigate the use or specialization of Axe for PFCS-based compositional proofs. Although PFCS aims at keeping proofs relatively small via parameterization and composition, Axe may come handy in case some large proof tasks arise. Axe's tactics may also be useful for certain proofs regardless of size, and not only for zero-knowledge circuits.

While interactive theorem proving is needed to verify parameterized circuit families with efficiency and generality, automated tools like SMT solvers could be useful for certain proof sub-tasks. ACL2 already has facilities to interface with automated reasoning tools.

There may be opportunities to partially automate the replication of the gadget constructions in the theorem prover, in the detached proof-generating extension approach (see Section 5.5). One avenue is the abstraction and translation of code in the gadget construction libraries. Another avenue, suggested by a reviewer, is to leverage any structure that can be recovered from generated gadgets.

9 Conclusion

Our exploration of the zero-knowledge circuit verification problem has shed more light into the problem, created and improved libraries and tools of more general use (e.g. the prime fields library), and evaluated increasingly sophisticated solution approaches. The PFCS-based compositional approach is promising, but completing the verification of the snarkVM gadgets will provide a more definitive validation.

The inherent restrictions on zero-knowledge circuits might initially lead to think of their verification as more tractable than general verification. Our exploration shows the opposite. As the size and complexity of the circuits grows, one eventually hits the "program verification wall". This should not be surprising, for a formalism that can describe sufficiently general computations. Although zero-knowledge circuits are not Turing-complete, and their verification is technically decidable because of their finiteness, the constraint solution space is so large that their verification is "practically undecidable".

Our exploration also confirms the importance of *structure* in formal verification. Preserving and leveraging the structure that is naturally available when the circuits are built promotes more manageable and efficient proofs, compared to losing that structure and then attempting to recover it.

Acknowledgements

Thanks to the Ethereum Foundation for funding Kestrel's work on the Semaphore circuit verification. Thanks to the Tezos Foundation for funding Kestrel's work on the Zcash circuit verification. Thanks

to the Zcash team for their help in extracting the Zcash R1CS constraints. Thanks to our colleagues Pranav Gaddamadugu and Collin Chin at Aleo for their help in extracting the snarkVM R1CS constraints. Thanks to the reviewers for comments that led to improvements to the paper.

References

[1] Aleo Systems Inc.: *The Aleo Blochchain*. Available at `https://aleo.org`.

[2] Aleo Systems Inc.: *The Leo Language*. Available at `https://leo-lang.org`.

[3] Aleo Systems Inc.: *snarkVM*. Available at `https://github.com/AleoHQ/snarkVM`.

[4] *The ark-r1cs-std Library*. Available at `https://github.com/arkworks-rs/r1cs-std`.

[5] Aztec: *The Noir Language*. Available at `https://aztec.network/noir`.

[6] *The bellman Library*. Available at `https://github.com/zkcrypto/bellman`.

[7] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh & Michael Riabzev (2018): *Scalable, transparent, and post-quantum secure computational integrity*. Cryptology ePrint Archive, Paper 2018/046. Available at `https://eprint.iacr.org/2018/046`.

[8] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer & Madars Virza (2014): *Zerocash: Decentralized Anonymous Payments from Bitcoin*. In: *Proc. 35th IEEE Symposium on Security and Privacy (SP)*, pp. 459–474, doi:10.1109/SP.2014.36.

[9] Nir Bitansky, Ran Canetti, Alessandro Chiesa & Eran Tromer (2011): *From Extractable Collision Resistance to Succinct Non-Interactive Arguments of Knowledge, and Back Again*. Cryptology ePrint Archive, Paper 2011/443. Available at `https://eprint.iacr.org/2011/443`.

[10] Manuel Blum, Paul Feldman & Silvio Micali (1988): *Non-interactive Zero-Knowledge and Its Applications*. In: *Proc. 20th Annual ACM Conference on Theory of Computing (STOC)*, pp. 103–112, doi:10.1145/62212.62222.

[11] Dan Bogdanov, Joosep Jääger, Peeter Laud, Härmel Nestra, Martin Pettai, Jaak Randmets, Ville Sokk, Kert Tali & Sandhra-Mirella Valdma (2022): *ZK-SecreC: a Domain-Specific Language for Zero Knowledge Proofs*, doi:10.48550/arXiv.2203.15448.

[12] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra & Howard Wu (2020): *ZEXE: Enabling Decentralized Private Computation*. In: *41st IEEE Symposium on Security and Privacy (SP)*, pp. 947–964, doi:10.1109/SP40000.2020.00050.

[13] Sean Bowe, Jack Grigg & Daira Hopwood (2019): *Recursive Proof Composition without a Trusted Setup*. Cryptology ePrint Archive, Paper 2019/1021. Available at `https://eprint.iacr.org/2019/1021`.

[14] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille & Greg Maxwell (2017): *Bulletproofs: Short Proofs for Confidential Transactions and More*. Cryptology ePrint Archive, Paper 2017/1066. Available at `https://eprint.iacr.org/2017/1066`.

[15] Vitalik Buterin (2016): *Quadratic Arithmetic Programs: from Zero to Hero*. Available at `https://medium.com/@VitalikButerin/quadratic-arithmetic-programs-from-zero-to-hero-f6d558cea649`.

[16] Collin Chin, Howard Wu, Raymond Chu, Alessandro Coglio, Eric McCarthy & Eric Smith (2021): *Leo: A Programming Language for Formally Verified, Zero-Knowledge Applications*. Cryptology ePrint Archive, Report 2021/651. Available at `https://eprint.iacr.org/2021/651`.

[17] Alessandro Coglio, Eric McCarthy, Eric Smith, Collin Chin, Pranav Gaddamadugu & Michel Dellepere (2023): *Compositional Formal Verification of Zero-Knowledge Circuits*. Cryptology ePrint Archive, Paper 2023/1278. Available at `https://eprint.iacr.org/2023/1278`. Presented at the Science of Blockchain Conference (SBC) 2023.

[18] Gilad Edelman (2021): *What is Web3, Anyway?* *WIRED*. Available at `https://www.wired.com/story/web3-gavin-wood-interview`.

[19] Electric Coin Company & Zcash Foundation: *The Zcash Blockchain*. Available at https://z.cash.

[20] Ethereum: *Zero-Knowledge Rollups*. Available at https://ethereum.org/en/developers/docs/scaling/zk-rollups.

[21] Cédric Fournet, Chantal Keller & Vincent Laporte (2016): *A Certified Compiler for Verifiable Computing*. In: *Proc. 29th IEEE Computer Security Foundations Symposium (CSF)*, pp. 268–280, doi:10.1109/CSF.2016.26.

[22] Shafi Goldwasser, Silvio Micali & Charles Rackoff (1985): *The Knowledge Complexity of Interactive Proof Systems*. In: *Proc. 17th Annual ACM Conference on Theory of Computing (STOC)*, pp. 291–304, doi:10.1145/22145.22178.

[23] Jens Groth (2016): *On the Size of Pairing-based Non-interactive Arguments*. Cryptology ePrint Archive, Paper 2016/260. Available at https://eprint.iacr.org/2016/260.

[24] Daira Hopwood, Sean Bowe, Taylor Hornby & Nathan Wilcox (2022): *Zcash Protocol Specification*. Available at https://github.com/zcash/zips/blob/master/protocol/protocol.pdf.

[25] Iden3: *The Circom Language*. Available at https://github.com/iden3/circom.

[26] Matt Kaufmann & J Strother Moore: *The ACL2 Theorem Prover*. Available at http://acl2.org.

[27] *The librustzcash Library*. Available at https://github.com/zcash/librustzcash.

[28] *The libsnark Library*. Available at https://github.com/scipr-lab/libsnark.

[29] Junrui Liu, Ian Kretz, Hanzhi Liu, Bryan Tan, Jonathan Wang, Yi Sun, Luke Pearson, Anders Miltner, Işıl Dillig & Yu Feng (2023): *Certifying Zero-Knowledge Circuits with Refinement Types*. Cryptology ePrint Archive, Report 2023/547. Available at https://eprint.iacr.org/2023/547.

[30] Daniel Lubarov & Brendan Farmer (2019): *R1CS Programming: ZK0x04 Workshop Notes*. Available at https://github.com/mir-protocol/r1cs-workshop/tree/master/workshop.pdf.

[31] Lurk Team: *The Lurk Language*. Available at https://lurk-lang.org.

[32] Ian Miers, Christina Garman, Matthew Green & Aviel D. Rubin (2013): *Zerocoin: Anonymous Distributed E-Cash from Bitcoin*. In: *Proc. 34th IEEE Symposium on Security and Privacy (SP)*, pp. 397–411, doi:10.1109/SP.2013.34.

[33] Alex Ozdemir, Fraser Brown & Riad S. Wahby (2022): *CirC: Compiler Infrastructure for Proof Systems, Software Verification, and More*. In: *Proc. 43rd IEEE Symposium on Security and Privacy (SP)*, pp. 2248–2266, doi:10.1109/SP46214.2022.9833782.

[34] Alex Ozdemir, Gereon Kremer, Cesare Tinelli & Clark Barrett (2023): *Satisfiability Modulo Finite Fields*. In: *Proc. 35th International Conference on Computer Aided Verification (CAV), Part II*, *Lecture Notes in Computer Science* 13965, pp. 163—-186, doi:10.1007/978-3-031-37703-7_8.

[35] Alex Ozdemir, Riad S. Wahby, Fraser Brown & Clark Barrett (2023): *Bounded Verification for Finite-Field-Blasting*. In: *Proc. 35th International Conference on Computer Aided Verification (CAV), Part III*, *Lecture Notes in Computer Science* 13966, pp. 154–175, doi:10.1007/978-3-031-37709-9_8.

[36] Shankara Pailoor, Yanju Chen, Franklyn Wang, Clara Rodríguez, Jacob Van Geffen, Jason Morton, Michael Chu, Brian Gu, Yu Feng & Işıl Dillig (2023): *Automated Detection of Under-Constrained Circuits in Zero-Knowledge Proofs*. *Proc. ACM Program. Lang.* 7(PLDI), doi:10.1145/3591282.

[37] Alex Pruden (2020): *How Zero-Knowledge is Rebalancing the Scales of the Internet*. Available at https://aleo.org/post/how-zero-knowledge-is-rebalancing-the-scales-of-the-internet.

[38] The ACL2 Community: *The ACL2 Theorem Prover and Community Books: Documentation*. Available at http:/acl2.org/manual.

[39] The ACL2 Community: *The ACL2 Theorem Prover and Community Books: Source Code*. Available at http://github.com/acl2/acl2.

[40] Franklyn Wang (2022): *Ecne: Automated Verification of ZK Circuits*. OxPARC Blog. Available at https://0xparc.org/blog/ecne.

Verification of GossipSub in ACL2s

Ankit Kumar Max von Hippel Panagiotis Manolios Cristina Nita-Rotaru
Northeastern University
Boston, USA
{kumar.anki, vonhippel.m, p.manolios, c.nitarotaru}@northeastern.edu

GossipSub is a popular new peer-to-peer network protocol designed to disseminate messages quickly and efficiently by allowing peers to forward the full content of messages only to a dynamically selected subset of their neighboring peers (mesh neighbors) while gossiping about messages they have seen with the rest. Peers decide which of their neighbors to graft or prune from their mesh locally and periodically using a score for each neighbor. Scores are calculated using a score function that depends on mesh-specific parameters, weights and counters relating to a peer's performance in the network. Since a GossipSub network's performance ultimately depends on the performance of its peers, an important question arises: Is the score calculation mechanism effective in weeding out non-performing or even intentionally misbehaving peers from meshes? We answered this question in the negative in our companion paper [31] by reasoning about GossipSub using our formal, official and executable ACL2s model. Based on our findings, we synthesized and simulated attacks against GossipSub which were confirmed by the developers of GossipSub, FileCoin, and Eth2.0, and publicly disclosed in MITRE CVE-2022-47547. In this paper, we present a detailed description of our model. We discuss design decisions, security properties of GossipSub, reasoning about the security properties in context of our model, attack generation and lessons we learnt when writing it.

1 Introduction

GossipSub is a new peer-to-peer network protocol used by popular applications like Eth2.0 [12] and FileCoin [6]. Messages transmitted in a GossipSub network are typically categorized into *topics*, which peers of the network can subscribe to or unsubscribe from. A peer can be part of several meshes corresponding to different topics. In contrast to *flood publishing* where a peer forwards every message it receives to all of its neighboring peers subscribed to the corresponding topic, GossipSub uses *lazy pull*, wherein a peer forwards full messages only to its *mesh neighbors* in the relevant topic. A peer can graft or prune a mesh neighbor based on various heuristic security mechanisms that ultimately rely on a locally computed score. The score is calculated periodically by each peer for each of its neighbors and is never shared. The score function, which is used to calculate a neighboring peer's score, depends on application-specific parameters and weights, and takes into account the performance of the neighbor both generally and on a given topic. Ideally, the score of misbehaving peers (*e.g.*, peers that drop messages or forward invalid ones) is penalized, which matters because negatively scored mesh neighbors get pruned.

The GossipSub developers specified their protocol in English prose [55, 56, 57] and implemented it in GoLang [4]. They relied on unit-tests and network emulation [58, 36] of pre-programmed scenarios for testing to show that misbehaving peers in a GossipSub network are eventually pruned. However, simple testing is not enough. Dijkstra famously quoted: "Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence."

In our companion paper [31], we formalized the GossipSub specification in the ACL2s (the ACL2 Sedan) [22, 13] theorem prover. ACL2s extends ACL2 [27, 28] with an advanced data definition framework (*Defdata*) [16], the *cgen* [17, 15, 18, 14] framework for automatic counterexample generation,

A. Coglio, S. Swords (Eds.): 18th ACL2 Workshop (ACL2-2023)
EPTCS 393, 2023, pp. 113–132, doi:10.4204/EPTCS.393.10

a powerful termination analysis based on calling context graphs [48] and ordinals [45, 46, 47], and a property-based modelling/analysis framework, each of which helped immensely in our formalization and verification effort. Our publicly available model [3] is designed to be modular and pluggable *i.e.*, it allows us to reason about parts of the model in isolation as well as about applications running on top of the network. Officially, GossipSub does not come with any properties. We formalized and attempted to prove security properties which (1) we thought should be reasonable for a score-based protocol like GossipSub to satisfy and which (2) the GossipSub developers agree with. One such property, which states that continuously misbehaving peers are eventually pruned, turned out to be invalid in the case of Eth2.0. We leveraged the cgen facility in ACL2s to automatically discover vulnerable network states that invalidate our property. We built attack gadgets using sequences of events which can take a network from a reasonable starting state to one of the discovered vulnerable states. We synthesized attacks which were confirmed by the GossipSub, FileCoin, and Eth2.0 developers and publicly disclosed in MITRE CVE-2022-47547. At the time of writing, the GossipSub developers are actively working on a fix. These results are explained in our companion paper [31]. In this work, we focus on our formalization of the GossipSub ACL2s model and its use for reasoning, simulation and attack synthesis.

Paper Outline. Section 2 describes the GossipSub protocol and our ACL2s model simultaneously. Section 3 describes properties we used to reason about GossipSub. Based on the insights gleaned from proving/disproving these properties, Section 4 describes how we synthesized attacks that can disrupt communication in an Eth2.0 GossipSub network. Section 5 describes some limitations of our model. Section 6 presents related work on mechanized theorem proving efforts in the field of distributed systems. Section 7 concludes.

2 GossipSub Model Description

In this section, we describe our ACL2s model, while also giving an overview of how the GossipSub protocol works. Interested readers are encouraged to walk through our ACL2s formalization whose presentation mirrors this section [2]. Consider the mesh shown in Figure 1. Full-message payloads are

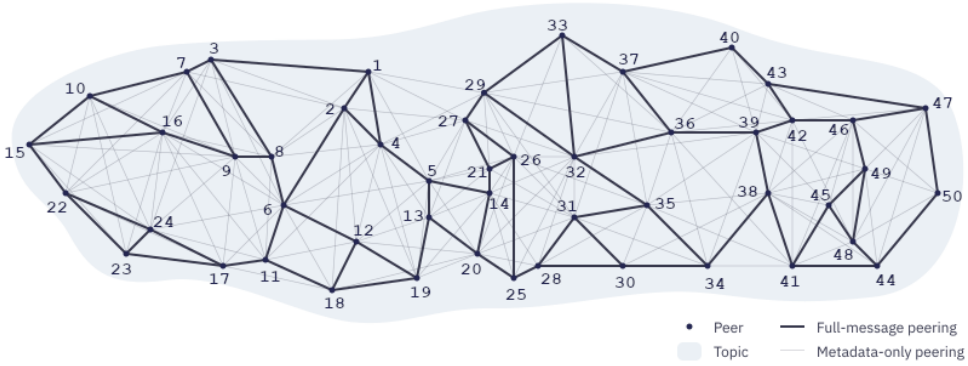


Figure 1: A mesh of peers subscribing to the same topic. This Figure was taken from [5].

forwarded on the full-message peering edges, which consume more bandwidth, while the metadata-only peering edges are used only to advertise and request full-messages using corresponding "`IHAVE`" and "`IWANT`" Remote Procedure Calls (RPCs). These RPCs carry the metadata of the full-message being advertised or requested, which is considerably smaller than the message itself. In this way, network

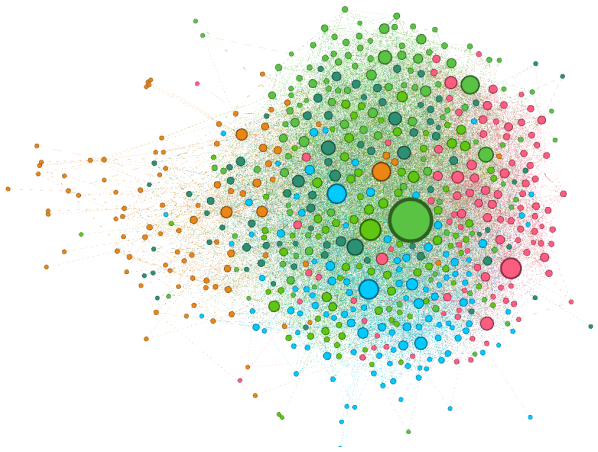


Figure 2: A community graph of Eth2.0 nodes where a bigger node implies greater degree. Similarly colored nodes have more edges among them than to the rest of the graph.

congestion is kept in check, and full-messages are supposed to be eventually disseminated to all peers who subscribe to the given topic. As an example, peers numbered `1` and `2` can send full-messages to each other, since they are mesh neighbors on the illustrated topic and share a full-message peering edge. However, peer `29` can receive a full-message from `1` only if it requests one, by sending an `IWANT` message in response to a corresponding `IHAVE` message received from peer `1`. Note, Figure 1 only shows a single GossipSub mesh. However, real world applications can have an arbitrary number of topics and corresponding meshes, *e.g.*, Eth2.0 supports up to 71 topics. Figure 2 illustrates a community graph of an actual Eth2.0 network from Li et. al. [35].

In order to model and analyze GossipSub in ACL2s, we rely heavily on Defdata to easily model network components, and on cgen for property-based testing. Distributed systems are often described in terms of *state machines*, namely automata that encode the discrete behaviors of a peer in the system [32, 53]. In our model, we describe the state-space of a GossipSub peer with a `peer-state` type, and then implement the GossipSub state machine using a state transition function. We use a Defdata map from peers to their corresponding states to capture the state of an entire network.

```
(defdata group (map peer peer-state))
```

We use records whenever we need to store state because of the convenience of using named fields to access the internals of the state. While proving theorems referring to states, we noticed that we routinely had to prove helper lemmas about the types of the contents of those states. This motivated us to rewrite records in the ACL2s books to enable such helper lemmas automatically, leading to cleaner code.

The local state of a peer is modeled as a record using the following definition:

```
(defdata peer-state
  (record (nts        . nbr-topic-state)
          (mst        . msgs-state)
          (nbr-tctrs  . pt-tctrs-map)
          (nbr-gctrs  . p-gctrs-map)
          (nbr-scores . peer-rational-map)))
```

where (1) `nts` is a record that stores information about the peer's neighbors' subscriptions, the peer's mesh neighbors, and the peer's fanout (which we describe later); (2) `mst` is a record that stores the peer's

messages and related state; (3) `nbr-tctrs` and `nbr-gctrs` are total maps that store counters used for computing a neighbor's topic-specific and general scores (respectively); and finally (4) `nbr-scores` is a total map from peers to their scores. Note that `peer` and `topic` are ACL2 symbols, while `tctrs` is a record that keeps track of a peer's behaviors in each topic:

```
(defdata tctrs
  (record (invalidMessageDeliveries . non-neg-rational)
          (meshMessageDeliveries    . non-neg-rational)
          (meshTime                 . non-neg-rational)
          (firstMessageDeliveries   . non-neg-rational)
          (meshFailurePenalty       . non-neg-rational)))
```

Similarly, `gctrs` keeps track of a peer's general behaviors, not pertaining to any single topic:

```
(defdata gctrs
  (record (apco . rational)  ;; application provided score
          (ipco . non-neg-rational) ;; ip-colocation factor
          (bhvo . non-neg-rational))) ;; track misbehavior
```

Notice that each of the counters is a rational, not a natural. This is because counters are supposed to fractionally decay at regular intervals. The rate of decay is dependent on the application running on top of GossipSub. We define maps for each of these counters:

```
(defdata pt (cons peer topic))
(defdata pt-tctrs-map (map pt tctrs))
(defdata p-gctrs-map (map peer gctrs))
(defdata peer-rational-map (map peer rational))
```

For each map, we define a lookup function with default values. ACL2s automatically proves termination and input-output contracts, which suffices to show that the maps are total. As an example, the following is the lookup function for scores:

```
(definecd lookup-score (p :peer prmap :peer-rational-map) :rational
  (let ((x (mget p prmap)))
    (match x
      (nil 0)  ;; default value
      (& x))))
```

We now explore each component of the `peer-state`.

Neighbor topics state. A key objective of the GossipSub protocol is to reduce network congestion, while forwarding data as quickly as possible. To achieve this, a peer forwards full messages only to a subset of its neighboring peers. A GossipSub peer keeps track of the topics its neighbors subscribe to, using the `nbr-topicsubs` field in `nbr-topic-state`. Using this information, it is able to build up a picture of the topics around it and which peers are subscribed to each topic. The peers to which it forwards full messages in topics it does not itself subscribe to constitute a list called a *fanout*, and is stored in the `topic-fanout` field, which is a map from topics to lists of peers (`lop`). The peer forwards full messages to other peers with whom it shares mesh membership. Mesh memberships are stored in the `topic-lop-map` field `topic-mesh`. Finally, the peer stores the last time it published a message to its fanout. This is used to expire a peer's fanout, if it has been too long since the peer last published.

```
(defdata peer symbol)
(defdata lop (listof peer))
(defdata topic-lop-map (map topic lop))
```

```
(defdata topic-nnr-map (map topic non-neg-rational))

(defdata nbr-topic-state
  (record (nbr-topicsubs . topic-lop-map)
          (topic-fanout  . topic-lop-map)
          (last-pub      . topic-nnr-map)
          (topic-mesh    . topic-lop-map)))
```

Messages state. `msgs-state` is a record type used to store information about messages received, requested, seen, or forwarded by a peer. First we define the types `pid-type`, which is just an alias for the type `symbol`, and the record type `payload-type`.

```
(defdata-alias pid-type symbol)
(defdata payload-type (record (content . symbol)
                              (pid     . pid-type)
                              (top     . topic)
                              (origin  . peer)))
```

`payload-type` is a record used to represent full messages, carrying the message content, payload id, the topic of this message and the peer who originated it. pid-type represents payload ids, a hash of a message payload which can identify the full message content. The msgs-state is defined as follows:

```
(defdata msg-peer (v (cons payload-type peer)
                     (cons pid-type peer)))
(defdata msgpeer-rat (map msg-peer rational))
(defdata msgs-waiting-for (map pid-type peer))
(defdata mcache (alistof payload-type peer))

(defdata msgs-state
  (record (recently-seen   . msgpeer-rat)
          (pld-cache       . mcache)
          (hwindows        . lon)
          (waitingfor      . msgs-waiting-for)
          (served          . msgpeer-rat)
          (ihaves-received . nat)
          (ihaves-sent     . nat)))
```

`msgs-state` stores (1) `recently-seen`, a map from either a full-message or a message hash to the time since receipt; (2) `pld-cache`, an association list of full messages and their senders; (3) `hwindows`, or history windows, a list of naturals where each is the number of messages received in an interval; (4) `waitingfor`, a map from message ids of messages that haven't been received yet, to peers one has sent corresponding `IWANT` requests to; and (5) `served`, a map from either a full-message or a message hash to a rational, denoting the count of the number of times this message was served. The `served` map helps to detect peers sending too many `IWANT` messages. Finally, `msgs-state` contains (6) `ihaves-received` and `ihaves-sent`, which store the number of `IHAVE` messages received and sent, respectively.

Events. Our model includes events that can occur in a network. Events include peers sending or receiving (1) control messages like `GRAFT` or `PRUNE` for mesh control, `SUB`, `UNSUB`, `JOIN` and `LEAVE` for topics, or `CONNECT1` or `CONNECT2` messages to edit neighbors; (2) `PAYLOAD` for carrying message payload, or `IHAVE` for advertising or `IWANT` for requesting message payloads; and (3) `HBM` for heart-beat events occurring at each peer at regular intervals. A list of events will have type `loev`. Events are defined as follows:

```
(defdata verb (enum '(SND RCV)))
(defdata rpc (v (list 'CONNECT1 lot)
                (list 'CONNECT2 lot)
                (list 'PRUNE topic)
                (list 'GRAFT topic)
                (list 'SUB topic)
                (list 'UNSUB topic)))
(defdata data (v (list 'IHAVE lopid)
                 (list 'IWANT lopid)
                 (list 'PAYLOAD payload-type)))
(defdata mssg (v rpc data))
(defdata evnt (v (cons peer (cons verb (cons peer mssg)))
                 (list peer 'JOIN topic)
                 (list peer 'LEAVE topic)
                 (list peer verb peer 'CONNECT1 lot)
                 (list peer verb peer 'CONNECT2 lot)
                 (list peer 'HBM pos-rational)
                 (list peer 'APP payload-type)))
(defdata hbm-evnt (list peer 'HBM pos-rational))
(defdata-subtype hbm-evnt evnt)

(defdata loev (listof evnt))
```

Heart-beat events occur at regular intervals at each peer in a GossipSub network. Several maintenance activities are performed during each such event. Scores are updated for neighboring peers, which are then used to update mesh memberships. Counters are multiplied by some application-specific decay factors. Neighboring peers that are not part of any mesh are sent `GRAFT` messages at regular intervals, provided that their addition could improve the average score of peers in the corresponding mesh. Several of these actions depend on application-specific weights (used for calculating scores) and parameters.

Parameters and Scoring. We store the weights and parameters relevant for scoring in a record called a `twp`. This record totally captures the application-specific configuration of a GossipSub instance, *e.g.*, we can uniquely specify the configuration used by Eth2.0 in a `twp`. Thus, in order to simulate an application running on top of GossipSub with our model, all we need to know is the application-specific `twp`. This is what we mean when we say our model is "pluggable".

```
(defdata weights
  (record (w1  . non-neg-rational)
          (w2  . non-neg-rational)
          (w3  . non-pos-rational)
          (w3b . non-pos-rational)
          (w4  . neg-rational)
          (w5  . non-neg-rational)
          (w6  . neg-rational)
          (w7  . neg-rational)))

(defdata params
  (record (activationWindow              . nat)
          (meshTimeQuantum               . pos)
          (p2cap                         . nat)
          (timeQuantaInMeshCap           . nat)
```

```
          (meshMessageDeliveriesCap       . pos-rational)
          (meshMessageDeliveriesThreshold . pos-rational)
          (topiccap                       . rational)
          (grayListThreshold              . rational)
          (d                              . nat)
          (dlow                           . nat)
          (dhigh                          . nat)
          (dlazy                          . nat)
          (hbmInterval                    . pos-rational)
          (fanoutTTL                      . pos-rational)
          (mcacheLen                      . pos)
          (mcacheGsp                      . non-neg-rational)
          (seenTTL                        . non-neg-rational)
          (opportunisticGraftThreshold    . non-neg-rational)
          (topicWeight                    . non-neg-rational)
          (meshMessageDeliveriesDecay     . frac)
          (firstMessageDeliveriesDecay    . frac)
          (behaviourPenaltyDecay          . frac)
          (meshFailurePenaltyDecay        . frac)
          (invalidMessageDeliveriesDecay  . frac)
          (decayToZero                    . frac)
          (decayInterval                  . pos-rational)))

(defdata wp (cons weights params))
(defdata twp (map topic wp))
```

Note that GossipSub required weights `w3` and `w3b` to be negative. However, the use of Defdata allowed us to automatically find that FileCoin used an invalid `twp` because it set `w3` and `w3b` to zero. We discussed this with the GossipSub developers who then agreed to allow zero values. Given `T`, a set of topics which the neighbor subscribes to; `tctrs`, the neighbor's topic specific counters; `gctrs`, the neighbor's global counters; and a `twp` containing entries for each topic our neighbor subscribes to, the score function calculates a neighbor's score as shown below.

$$score(q) = \min\left(\mathrm{TC}, \sum_{\tau \in T} \mathrm{tw}^{\tau} \times \sum_{i \in \{1,2,3,3b,4\}} w_i^{\tau} P_i^{\tau}\right) + w_5 P_5 + w_6 P_6 + w_7 P_7$$

where

$$\texttt{(weights . params)} = \texttt{(mget } \tau \texttt{ twp)}$$
$$w_i^{\tau} = \texttt{(weights-w}i\texttt{ weights)}$$
$$P_1^{\tau} = \texttt{(calcP1(tctrs-meshTime tctrs) (params-meshTimeQuantum params)}$$
$$\texttt{(params-timeQuantaInMeshCap params))}$$
$$P_2^{\tau} = \texttt{(calcP2(tctrs-firstMessageDeliveries tctrs) (params-p2cap params))}$$
$$P_3^{\tau} = \texttt{(calcP3(tctrs-meshTime tctrs) (params-activationWindow params)}$$
$$\texttt{(tctrs-meshMessageDeliveries tctrs)}$$
$$\texttt{(params-meshMessageDeliveriesCap params)}$$
$$\texttt{(params-meshMessageDeliveriesThreshold params))}$$
$$P_{3b}^{\tau} = \texttt{(calcP3b(tctrs-meshTime tctrs) (params-activationWindow params)}$$

(tctrs-meshFailurePenalty tctrs)
(tctrs-meshMessageDeliveries tctrs)
(params-meshMessageDeliveriesCap params)
(params-meshMessageDeliveriesThreshold params))

P_4^τ = (calcP4(tctrs-invalidMessageDeliveries tctrs))

P_5 = (gctrs-apco gctrs)

P_6 = (gctrs-ipco gctrs)

P_7 = (calcP7 (gctrs-bhvo gctrs))

tw^τ = (params-topicweight params)

TC = (params-topiccap (cddar twp))

TC does not depend on any topic, but since it is stored in a `twp` which is indexed by `topic`, its value is replicated in each of the corresponding `params`. So, it is fine to extract its value from the first entry of a `twp`. Note that for score calculations, we require a non-empty `twp`.

Each of the `calcPi` functions where `i` $\in \{$`1,2,3,3b,4,7`$\}$ is used to calculate contributions to the score by one or more of counter values from `tctrs`. `calcP1` calculates the contribution to a neighbor's score based on the time spent in common meshes. `calcP2` awards score for being one of the first few to forward a message. `calcP3` calculates penalties due to the mesh message transmission rate being below a given threshold of `(params-meshMessageDeliveriesThreshold params)`. Whenever a peer is pruned, its corresponding `tctrs` counter `meshFailurePenalty` is augmented by the mesh message transmission rate deficit. This counter is not cleared even after the peer has been pruned. `calcP3b` scores mesh message delivery failures based on the value of this counter. Hence, P_{3b}^τ is a "sticky" value which is supposed to discourage a peer that was pruned because of under-delivery from quickly getting re-grafted in a mesh. `calcP4` calculates the penalty on score due to sending invalid messages. `calcP7` calculates penalties due to several kinds of misbehaviors described by the GossipSub specification. An example of such misbehaviors includes spamming with too many `IHAVE` messages which are either bogus and/or not following up to the corresponding `IWANT` requests.

The Transition Function. We define a transition function `run-network`, which, given an initial `Group` state and a list of `evnt`, produces a trace of type `egl`, which is an alist of `evnt` and `group`. Hence, after running a simulation, we have access to the state of the `Group` after each `evnt` was processed. `run-network` depends on the transition function for the `peer-state` (`transition`), which depends on the transition functions for `nbr-topic-state` (`update-nbr-topic-state`) and for `msgs-state` (`update-msgs-state`). For brevity, we mention only the signatures of each of these functions below. Notice that each of these signatures represents neatly and concisely the types of the formal arguments and the function return type, which is very useful in a large code base.

```
(defdata egl (alistof evnt group)) ;; simulation trace

(definecd run-network (gr :group evnts :loev i :nat r :twp s :nat) :egl
...)

(defdata peer-state-ret
  (record (pst . peer-state)
          (evs . loev)))

(definecd transition
```

```
  (self :peer pstate :peer-state evnt :evnt r :twp s :nat) :peer-state-ret
...)

(defdata msgs-state-ret
  (record (mst . msgs-state)
          (evs . loev)
          (tcm . pt-tctrs-map)
          (gcm . p-gctrs-map)))

(definecd update-msgs-state (mst :msgs-state evnt :evnt pcm :pt-tctrs-map
                             gcm :p-gctrs-map r :twp) :msgs-state-ret
...)

(defdata nbr-topic-state-ret
  (record (nts . nbr-topic-state)
          (evs . loev)
          (tcm . pt-tctrs-map)
          (gcm . p-gctrs-map)
          (sc  . peer-rational-map)))

(definecd update-nbr-topic-state (nts :nbr-topic-state
                                  nbr-scores :peer-rational-map
                                  tcm :pt-tctrs-map gcm :p-gctrs-map
                                  evnt :evnt r :twp s :nat) :nbr-topic-state-ret
...)
```

A GossipSub peer can select a random subset of its fanout and promote them as mesh members. It can also selects a random subset of its neighbors to advertise with "IHAVE" messages. Such non-determinism is handled by sending a random seed `s` as a formal parameter to `run-network`, which is then propagated to the other transition functions it depends on. Observe that a single event like full message forwarding can trigger several more forwards, causing a cascade of events. Such events are represented by the `evs` field in the return types of `update-msgs-state` and `update-nbr-topic-state`. In order to limit the total number of events processed, we send a natural number `i` as a formal parameter to the `run-network` function.

When proving contract theorems for the transition functions, we needed to prove the types of terms returned by utility functions, like `shuffle`, or ACL2 functions like `set-difference-equal`. For this, we used polymorphism and automated type-based reasoning provided by Defdata, as shown below:

```
(sig set-difference-equal ((listof :a) (listof :a)) => (listof :a))
(sig shuffle ((listof :a) nat) => (listof :a))
```

We made heavy use of higher-order macros written by Manolios [1] to improve the readability of our code. For example, `create-map*` is a list functor. Given an admitted function name or a lambda expression `f` of type `a` $\rightarrow$ `b`, `create-map*` defines a function `map*-*f` of type `(listof a)` $\rightarrow$ `(listof b)`. In the following code snippet, we show theorems proving that it obeys the functor laws, and give an example of its usage. `map*` is a syntactic sugar that maps function `f` onto a list without referring to the generated function name `map*-*f`.

```
(definec id (x :all) :all
  x)
;; Proof that create -map* is a list functor
;; 1) functor mapping preserves the identity function
```

```
(property functor-id (xs :tl)
  (== (map* id xs) ;; id is an identity function for lists
      (id xs)))

;; 2) functor mapping preserves function composition. f and g are declared
;; using defstub. gof is defined as a composition of g and f
(property functor-comp (xs :tl)
  (== (map* gof xs)
      (map* g (map* f xs))))

;; function to create a list of SND GRAFT events from peer p to a list of peers
(create-map* (lambda (tp p) `(,p SND ,(cdr tp) GRAFT ,(car tp)))
             lotopicpeerp
             loevp
             (:name mk-grafts)
             (:fixed-vars ((peerp p))))

(check= (map* mk-grafts '((FM . A) (DS . B)) 'P)
        '((P SND A GRAFT FM) (P SND B GRAFT DS)))
```

Given an admitted function name or a lambda expression `f` of type `a` $\times$ `b` $\rightarrow$ `b`, the higher order function `create-reduce*` defines a function `reduce*-*f` which accepts a list of elements of type `a`, an initial accumulator value of type `b`, and returns a reduction of the list using `f`, from left to right.

```
;; function to extract all the subscribers (neighboring peers) from a topic-lop-map
(create-reduce* (lambda (tp-ps tmp) (app tmp (cdr tp-ps)))
                lopp
                topic-lop-mapp
                (:name subscribers))

(check= (reduce* subscribers '()
                 '((T1 P1 P2 P3)
                   (T2 P4 P5 P1)))
        '(P4 P5 P1 P1 P2 P3))
```

3 Reasoning about the scoring function

Based on the observation that honest peers can be distinguished from malicious ones based on their observable behaviors (using local counters and scores), and thus, the overall network can be made more secure and performant if every honest peer promotes their well-behaving neighbors and demotes poorly-behaved ones, we came up with the following informal fundamental property.

Fundamental Property of GossipSub Defense Mechanisms. *Peers who behave poorly will be demoted by their neighbors. Peers who behave better-than-average will be promoted by their neighbors. Promotion/demotion is entirely based on local peer behavior.*

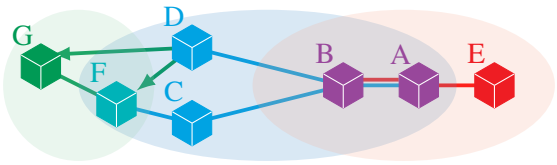


Figure 3: An example network.

Before reasoning about the formalization of this fundamental property later in the paper, we discuss the importance of this fundamental property. Consider the simple example shown in Figure 3, where peers A and B subscribe to, and are mesh neighbors in both Red and Blue topics. It might be possible for B to observe that A behaves perfectly well in the Blue topic while simultaneously misbehaving in the Red topic. This is not good for B because it depends on A for all of its messages in the Red topic. Note that this is a very simplified example. In an actual network, B could have several other neighbors subscribed to the Red topic. But as we will show later, it is equally trivial to have a scenario where all of B's neighbors isolate it from communications in the Red topic. In this example, we want B to prune A from its Red mesh in hopes of finding a better mesh neighbor later on. Reasoning about this fundamental property directly would be difficult due to the massive search-space of possible attack vectors. Hence, we focus on the following liveness property capturing the essence of the fundamental property:

Property 1 If a peer's score relating to its performance in any topic is continuously non-positive, then the peer's overall score should eventually be non-positive:

$$\forall q, \tau :: \langle \boldsymbol{G}(score(q) \textit{ for topic } \tau \leq 0) \Rightarrow \boldsymbol{F}(score(q) \leq 0) \rangle$$

where $score(q)$ *for topic* t *is defined below.*

$$\mathrm{tw}^{\tau} \times \sum_{i \in \{1,2,3,3b,4\}} w_i^{\tau} P_i^{\tau}$$

Notice that Property 1 is temporal. We write the non-temporal version of this property in context of Eth2.0 (using Eth2.0 `twp`) in ACL2s as shown below, and disprove the temporal version using an induction argument later in the paper.

Property 2

```
(property (ptc :pt-tctrs-map pcm :p-gctrs-map p :peer top :topic)
  :hyps  (^ (member-equal '(,p . ,top) (acl2::alist-keys ptc))
            (> (lookup-score p (calc-nbr-scores-map ptc pcm *eth-twp*)) 0))
  (> (calcScoreTopic (lookup-tctrs p top ptc) (mget top *eth-twp*)) 0))
```

The following is one of the counter-examples to the above property, generated by cgen in ACL2s:

```
((top 'agg)
 (p 'p4)
 (pcm '((p3449 (:0tag . gctrs) (:apco . 0) (:bhvo . 0) (:ipco . 0))
        (p3450 (:0tag . gctrs) (:apco . 0) (:bhvo . 0) (:ipco . 0))
        (p3451 (:0tag . gctrs) (:apco . 0) (:bhvo . 0) (:ipco . 0))))
 (ptc '(((p4 . agg)
         (:0tag . tctrs)
         (:firstmessagedeliveries . 0)
         (:invalidmessagedeliveries . 0)
```

```
        (:meshfailurepenalty . 0)
        (:meshmessagedeliveries . 1)
        (:meshtime . 42))
       ((p4 . blocks)
        (:0tag . tctrs)
        (:firstmessagedeliveries . 324)
        (:invalidmessagedeliveries . 0)
        (:meshfailurepenalty . 0)
        (:meshmessagedeliveries . 330)
        (:meshtime . 377))
       ((p4 . sub1)
        (:0tag . tctrs)
        (:firstmessagedeliveries . 371)
        (:invalidmessagedeliveries . 0)
        (:meshfailurepenalty . 0)
        (:meshmessagedeliveries . 377)
        (:meshtime . 324))
       ((p4 . sub2)
        (:0tag . tctrs)
        (:firstmessagedeliveries . 318)
        (:invalidmessagedeliveries . 0)
        (:meshfailurepenalty . 0)
        (:meshmessagedeliveries . 324)
        (:meshtime . 371))
        ... ))
```

For brevity, we omit entries for peer-topic key values for peers other than `p4`. In property-based testing, the free variables of a property under test are assigned values using a synergistic combination of theorem proving and random assignments computed using type-based enumerators (generators) in an effort to discover counterexamples to the property. Observe that Property 2 depends on `ptc` and `pcm` which do not have trivial types. These are maps containing records which themselves consist of several numerical values, which makes the search space of possible counter-examples immensely large. In order to make it easier for cgen to find counter-examples, we wrote custom enumerators to enumerate restricted values for `topic`, `tctrs`, `gctrs`, `pt-tctrs-map` and `p-gctrs-map`. Specifically, we limit the penalties on the score due to some counters, such that the negative contributions due to misbehavior are comparable to the positive contributions due to good behavior. Below we define custom enumerators for `topic` and `tctrs`.

```
(definec topics () :tl
  ;; valid topics used in Ethereum
  '(AGG BLOCKS SUB1 SUB2 SUB3))

(definec nth-topic-custom (n :nat) :symbol
  (nth (mod n (len (topics))) (topics)))
(defdata lows (range integer (0 <= _ <= 1))) ;; high values
(defdata-subtype lows nat)
(defdata highs (range integer (300 < _ <= 400))) ;; low values
(defdata-subtype highs nat)

(defun nth-bad-counters-custom (n)
;; setting invalidMessageDeliveries and meshFailurePenalty to 0 due to high penalty
```

```
(defun nth-good-counters-custom (n)
  (tctrs 0 (nth-highs (+ n 2)) (nth-highs (+ n 3)) (nth-highs (+ n 4)) 0))

(defun nth-counters-custom (n) ;; custom enumerator for tctrs
  (if (== 0 (mod n 4))
      (nth-bad-counters-custom n)
    (nth-good-counters-custom n)))
```

Besides Property 2, we formalize three safety properties for GossipSub, stated below. Together, these four are the most general properties of the score function, which must hold in order for the fundamental property to hold.

Property 3 Increasing bad-performance counters (which are multiplied with negative weights) should decrease the overall score.

Property 4 Increasing good-performance counters (which are multiplied with positive weights) will not decrease the score for a mesh peer that has been in the mesh for a sufficiently long time.

Property 5 If two peers subscribe to the same topics, and achieve identical per-topic counters, and identical global counters, then they achieve identical scores.

We are able to find counterexamples to Property 3 in much the same way as 2, however, in this work we focus on the counterexamples to Property 2, which are more interesting in terms of attack generation. We manually prove that Property 4 holds over all configurations (available with the paper artifacts). The proof that Property 5 holds over all configurations follows directly from referential transparency.

We also prove a limit on the maximum score achievable in a topic, as shown below.

```
(property max-topic-score (tctrs :tctrs weights :weights params :params)
  (<= (calcScoreTopic tctrs (cons weights params))
      (* (params-topicweight params)
         (+ (* (mget :w1 weights) (params-timeQuantaInMeshCap params))
            (* (mget :w2 weights) (params-p2cap params))))))
```

4 Attack Generation

The counter-example 3 which we obtained in the previous section is a specification for an unsafe state that does not satisfy Property 2 for an Eth2.0 network. However, we are also interested in characterizing and generating attacks against an Eth2.0 GossipSub network for three main reasons: (1) to show that an unsafe state is **reachable** from a reasonable start state, (2) **invalidation** of our temporal Property 1 using the trace generated from the attack, and finally (3) demonstration of **scalability** of our attack to large networks, typically of the size and shape used by real world applications.

Counter-example 3 suggests that an unsafe state is one where a neighboring peer throttles communication in a particular topic while maintaining an overall positive score, hence, avoiding getting pruned. Using this insight, we design attack gadgets that can perpetrate such attacks locally. We define an *attack gadget* as a tuple $\langle A, V, S \rangle$, where A, V are peers (A is the attacker and V is the victim), S is a set of subnet topics (the attacked topics), and A, V are mesh neighbors over a set of topics that is a superset of S. For each $i \in \mathbb{N}$, we define AG_i to be the set of attack gadgets where $|S| = i$. Figure 4 illustrates an example attack gadget in AG_2 where peers A and V are neighbors in four meshes corresponding to topics: Red, Yellow, Blue and Green, out of which A is attacking V in $S = \{\text{Red}, \text{Blue}\}$. We generate a sequence of

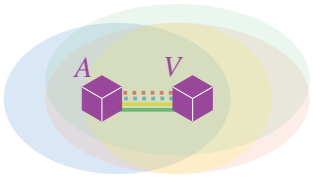


Figure 4: An example AG_2 attack gadget $\langle A, V, \{\text{Red}, \text{Blue}\}\rangle$.

events consisting of message transmission events from A to V (referred to as `a` and `v` in code) as well as heart-beat events at V (V updates the scores of its peers during heart–beat events). These events are designed to either restrict or completely block communication to V in the attacked topics while maintaining normal communication rate in all the other topics, as shown in the following code snippet.

```
(definecd emit-evnts (a v :peer ts ats :lot n m e :nat) :loev
  ;; mesh message deliveries in attacked topics
  (app (emit-meshmsgdeliveries-peer-topics a v ats m)
       ;; mesh message deliveries in other topics
       (emit-meshmsgdeliveries-peer-topics a v (set-difference-equal ts ats) n)
       ;; heart-beat events at the victim node
       '((,p2 HBM ,e))))
```

The expression `(emit-meshmsgdeliveries-peer-topics a v ts ats n m e)` generates a list of events E sending `m` mesh messages in the attacked topics and `n` mesh messages in the other topics from peer `a` to peer `v` per heart-beat at `v` which happens every `e` seconds. `m` is generally set to `0` or `1` in order to block or throttle communication in the attacked topic meshes. We choose a start state of the network based on actual full-network topologies of the Eth2.0 testnets Ropsten (shown in Figure 2), Goerli and Rinkeby, as measured by Li et. al [35]. Table 1 characterizes each of these topologies.

Network	Nodes	Degree			Diameter
		min	max	avg	
Ropsten	588	1	418	25.49	5
Goerli	1355	1	712	28.26	5
Rinkeby	446	1	191	68.96	6

Table 1: Eth2.0 Network Characteristics

We create our start state as a `Group`, using the topologies provided. In this `Group`, we initialize our attack gadget and simulate its run over the generated sequence of events E using the `run-network` function. We use function `scorePropViolation` described below, to detect violations of Property 2 in peer states occurring in the output trace.

```
;; ats is the list of topics being attacked
(definec scorePropViolation (ps :peer-state p :peer ats :lot twpm :twp) :boolean
  (match ats
    (() (> (lookup-score p (calc-nbr-scores-map (peer-state-nbr-tctrs ps)
                                                (peer-state-nbr-gctrs ps) twpm))
           0))
    ((top . rst) (^ (< (calcScoreTopic
```

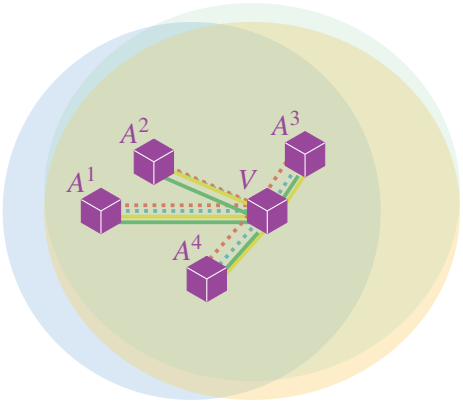


Figure 5: An eclipse attack using AG_2 gadgets

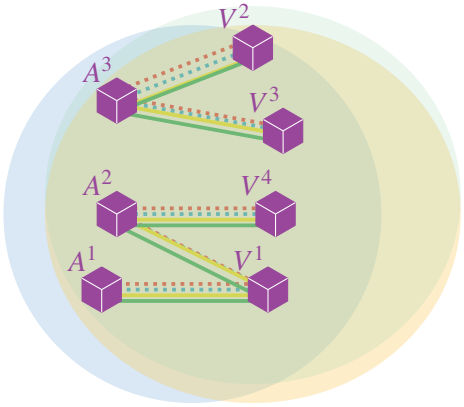


Figure 6: A partition attack using AG_2 gadgets

```
                (lookup-tctrs p top (peer-state-nbr-tctrs ps))
                (mget top twpm))
               0)
        (scorePropViolation ps p rst twpm)))))
```

We observe that after the first heart-beat event at the victim peer, the state of the network at each subsequent heart-beat is identical. Since Property 2 is being violated at each of these events, we make an inductive claim that it will forever be violated, thus giving a counter-example to our liveness Property 1. We wrote optimized versions of the `run-network` function to generate traces of property violations as a list of booleans, instead of generating the full trace of type `egl`. We ran our simulations for 100,000 events to ensure that we ended up in identical states, taking about a minute on an M1 Macbook Air.

Finally, we scaled our attacks to build *eclipse* and *network partition* attacks using a combination of attack gadgets. Figures 5 and 6 give an intuition of how to combine our attack gadgets to carry out these attacks. Our companion paper discusses the specifics of these attacks in more detail.

5 Limitations

We now discuss limitations of our model. The most crucial aspect of property-based testing is counter-example generation for invalid properties. As explained previously, our properties depend on complex types, for which we had to write custom enumerators. Coming up with enumerators that had a high probability of satisfying the hypotheses of our properties required considerable analysis of Eth2.0 so as to restrict certain `tctrs` values from skewing the scores too much. Testing properties for new applications will likewise require writing new custom enumerators. One might need to write a new event generator as well, possibly generating events of shapes different from the ones we showed.

6 Related Work

The Protocol Labs ResNetLab and software audit firm Least Authority tested GossipSub against a list of specific pre-programmed attack scenarios [59] designed to degrade overall network performance using a network emulator called TESTGROUND [7]. Due to their use of simplified configurations with only one topic (and because simple testing is not enough to find bugs) they found that all the attacks failed against

GossipSub, and the score function made GossipSub more resilient to malicious nodes attacks than the other tested protocols [58]. Separate from simulation testing, Least Authority also audited the Golang implementation and provided recommendations for improvement [36]. Though our work contributes the first formalization of GossipSub, there has been considerable previous work on utilizing formal methods to reason about distributed systems. We survey such works below.

Model Checking based approaches. Lamport's modeling language TLA+ [33] and the corresponding TLC model checker [63] have been used to analyze properties of distributed systems including DISK PAXOS [23], MONGORAFTRECONFIG [54], Byzantine PAXOS [34], SPIRE [30], etc. McMillan and Zuck applied specification-based testing to the QUIC protocol, and found vulnerabilities [49]. Wu et. al. formally modeled the Bluetooth stack using PROVERIF, a model checker, and found five known vulnerabilities and two new ones [62]. Chothia et. al. demonstrated the use of PROVERIF to verify distance-bounding protocols, *e.g.*, those used by MasterCard and NXP [20]. Separately, Chothia modeled the MUTE anonymous file-sharing system using the π-calculus, and proved the system insecure (discovering a novel attack) [19]. Cremers et. al. modeled all handshake modes of TLS 1.3 using TAMARIN, another model checking tool, and discovered an unexpected behavior [21]. An issue with using model checking tools like ProVerif or Tamarin to verify a protocol like GossipSub is the immense size of the state space needed to be checked, making them infeasible for our use.

Refinement-based proof formalization. The theory of refinement has proved to be useful for enabling the mechanical verification of distributed systems' properties. Manolios' work on refinement [38, 40, 39] has been previously used for mechanical verification of pipelined processors [37, 41, 42, 44, 43]. Manolios et. al. combined theorem proving (using refinement maps) with model checking to verify the alternating bit protocol. [40]. IRONFLEET [25] refines TLA style state-machine specification of a PAXOS-based library and a sharded key-value store to low level implementation in Dafny (a SMT based program verifier) for verification using Hoare-logic. Woo et. al. [61] formally verified 90 properties of the RAFT protocol using VERDI [60], a tool they built in the COQ proof assistant. Though they did not build an executable model, their framework can be used to extract an executable protocol implementation in OCaml. VERDI provides verified *system transformers* used to refine a system in an ideal fault model to a more realistic fault model, without any proof overhead on part of the user. We believe that looking at GossipSub through the lens of refinement will be interesting because, not only will it allow us to explain why it failed our properties, but also guide us towards improving it.

Inductive-invariant based proof formalization. Padon et. al. [51] proved the correctness of a simple model of Paxos described in Effectively Propositional Logic (a decidable fragment of First Order Logic) using IVY [52], a SMT-based safety verification tool. IVY can be used for verifying inductive invariants about global states of a distributed protocol. Both the modeling and the specification languages of IVY are restricted to a decidable fragment of First Order Logic to ensure that all verification conditions can be checked algorithmically. Hippel et. al. [26] also used IVY to formally describe and reason about Karn's Algorithm, a mechanism used to study rount trip times of message transmissions. However, since IVY lacks a theory of rationals, modelling the scoring function of GossipSub would not have been possible using this tool.

Full stack verification. Certain high-assurance distributed systems might require the whole stack to be formally-verified. Such applications could, for instance, be implemented on top of SEL4: a high-performance operating system microkernel that was formally verified against an abstract specification using higher-order logic [29]. Another example is the fully verified *CLI stack* [8], a system comprising of an operating system with some applications running in it, operational semantics for two high level languages, a stack based assembly language and the instruction set architecture (ISA), all the way down to the register transfer level (RTL) design for a microprocessor. The full stack was verified in Nqthm [11].

7 Conclusion and Future Work

In this paper, we described the GossipSub protocol, as well as our official formalization based on its prose specification using the ACL2s theorem prover. We explained our state models, transition functions as well as design decisions. We showed our security property for GossipSub and how we were able to find counter-examples against it. Finally, we described several kinds of attacks we synthesized based on our attack gadgets, using the couter-examples as specifications.

In the future, we would like to characterize an ideal variant of GossipSub as a refinement of simpler protocols so as to prove safety properties, as well as to contrast the ideal variant with our current model in order to better explain why it is susceptible to attacks from misbehaving peers. We would also like to support reasoning for the application layer on top of our network model layer, since interesting bugs can be found at the interface of these two layers.

References

[1] *ACL2 Sources*. https://github.com/acl2/acl2. Accessed 12 June 2023.

[2] *Code walk of the GossipSub ACL2s formalization*. https://github.com/maxvonhippel/gossipSub-FM/blob/main/model/demo.lisp. Accessed 30 May 2023.

[3] *GossipSub Model and attacks*. https://github.com/ankitku/gsacl2ws. Accessed 22 July 2023.

[4] GO-LIBP2P-PUBSUB. https://github.com/libp2p/go-libp2p-pubsub.

[5] *What is Publish/Subscribe*. https://docs.libp2p.io/concepts/pubsub/overview/. Accessed 12 May 2023.

[6] (2017): *Filecoin: A Decentralized Storage Network*. https://filecoin.io/filecoin.pdf.

[7] (2022): TESTGROUND. https://docs.testground.ai/. Accessed 24 July 2022.

[8] William Bevier, Warren Hunt, Jstrother Moore & William Young (1989): *Special issue on system verification*. *Journal of Automated Reasoning*.

[9] Bruno Blanchet (2016): *Modeling and verifying security protocols with the applied pi calculus and ProVerif*. *Foundations and Trends® in Privacy and Security*, doi:10.1561/3300000004.

[10] Paul Bonsma (2010): *Most balanced minimum cuts*. *Discrete Applied Mathematics*, doi:10.1016/j.dam.2009.09.010.

[11] R.S. Boyer, M. Kaufmann & J.S. Moore (1995): *The Boyer-Moore theorem prover and its interactive enhancement*. *Computers and Mathematics with Applications*, doi:10.1016/0898-1221(94)00215-7. Available at https://www.sciencedirect.com/science/article/pii/0898122194002157.

[12] Vitalik Buterin (2014): *Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform*. https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum_Whitepaper_-_Buterin_2014.pdf. Accessed 13 July 2022.

[13] Harsh Chamarthi, Peter C. Dillinger, Panagiotis Manolios & Daron Vroon (2011): *The "ACL2" Sedan Theorem Proving System*. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, doi:10.1007/978-3-642-19835-9_27.

[14] Harsh Raju Chamarthi (2016): *Interactive Non-theorem Disproving*. Ph.D. thesis, Northeastern University, doi:10.17760/D20467205.

[15] Harsh Raju Chamarthi, Dillinger Peter C., Matt Kaufmann & Panagiotis Manolios (2011): *Integrating testing and interactive theorem proving*. doi:10.4204/EPTCS.70.1.

[16] Harsh Raju Chamarthi, Dillinger Peter C. & Panagiotis Manolios (2014): *Data Definitions in the ACL2 Sedan*. *ACL2*.

[17] Harsh Raju Chamarthi, Peter C. Dillinger, Matt Kaufmann & Panagiotis Manolios (2011): *Integrating Testing and Interactive Theorem Proving*. In David S. Hardin & Julien Schmaltz, editors: *Proceedings 10th International Workshop on the ACL2 Theorem Prover and its Applications*, *EPTCS* 70, pp. 4–19, doi:10.4204/EPTCS.70.1.

[18] Harsh Raju Chamarthi & Panagiotis Manolios (2011): *Automated specification analysis using an interactive theorem prover*. In Per Bjesse & Anna Slobodová, editors: *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11*, FMCAD Inc., pp. 46–53. Available at `http://dl.acm.org/citation.cfm?id=2157665`.

[19] Tom Chothia (2006): *Analysing the MUTE anonymous file-sharing system using the pi-calculus*. In: *International Conference on Formal Techniques for Networked and Distributed Systems*, doi:10.1007/11888116_9.

[20] Tom Chothia, Joeri De Ruiter & Ben Smyth (2018): *Modelling and analysis of a hierarchy of distance bounding attacks*. In: *27th USENIX Security Symposium (USENIX Security 18)*.

[21] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott & Thyla van der Merwe (2017): *A Comprehensive Symbolic Analysis of TLS 1.3*. In: *Conference on Computer and Communications Security*, doi:10.1145/3133956.3134063.

[22] Peter C. Dillinger, Panagiotis Manolios, Daron Vroon & J Strother Moore (2007): *ACL2s: "The ACL2 Sedan"*. In: *International Conference on Software Engineering (ICSE)*, doi:10.1109/ICSECOMPANION.2007.14.

[23] Eli Gafni & Leslie Lamport (2003): *Disk paxos*. *Distributed Computing*, doi:10.1007/s00446-002-0070-8.

[24] David A. Greve, Matt Kaufmann, Panagiotis Manolios, J. Strother Moore, Sandip Ray, José-Luis Ruiz-Reina, Robert W. Sumners, Daron Vroon & Matthew Wilding (2008): *Efficient execution in an automated reasoning environment*. *J. Funct. Program.*, doi:10.1017/S0956796807006338.

[25] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty & Brian Zill (2015): *IronFleet: proving practical distributed systems correct*. In: *Proceedings of the 25th Symposium on Operating Systems Principles*, doi:10.1145/2815400.2815428.

[26] Max von Hippel, Kenneth L. McMillan, Christina Nita-Rotaru & Lenore D. Zuck (2023): *A Formal Analysis of Karn's Algorithm*. In: *Networked Systems*, doi:10.1007/978-3-031-37765-5_4.

[27] Matt Kaufmann & J Strother Moore (1996): *ACL2: An industrial strength version of Nqthm*. In: *Proceedings of 11th Annual Conference on Computer Assurance (COMPASS)*, doi:10.1109/CMPASS.1996.507872.

[28] Matt Kaufmann & J Strother Moore (2022): *ACL2 homepage*. Available at `https://www.cs.utexas.edu/users/moore/acl2/`.

[29] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski & Michael Norrish (2009): *seL4: Formal verification of an OS kernel*. In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, doi:10.1145/1629575.1629596.

[30] Emil Koutanov (2021): *Spire: A Cooperative, Phase-Symmetric Solution to Distributed Consensus*. *IEEE Access*, doi:10.1109/ACCESS.2021.3096326.

[31] Ankit Kumar, Max von Hippel, Pete Manolios & Cristina Nita-Rotaru (2022): *Formal Model-Driven Analysis of Resilience of GossipSub to Attacks from Misbehaving Peers*. *arXiv preprint arXiv:2212.05197*, doi:10.48550/arXiv.2212.05197.

[32] Leslie Lamport (1978): *The implementation of reliable distributed multiprocess systems*. *Computer Networks*, doi:10.1016/0376-5075(78)90045-4.

[33] Leslie Lamport (2002): *Specifying systems: the TLA+ language and tools for hardware and software engineers*.

[34] Leslie Lamport (2011): *Byzantizing Paxos by refinement*. In: *International symposium on distributed computing*, doi:10.1007/978-3-642-24100-0_22. TLA+ proof available at `https://lamport.azurewebsites.net/tla/byzpaxos.html`, accessed 29 July 2022.

[35] Kai Li, Yuzhe Tang, Jiaqi Chen, Yibo Wang & Xianghong Liu (2021): *TopoShot*. In: *Internet Measurement Conference*, doi:10.1145/3487552.3487814.

[36] Dylan Lott (2020): *Audit of Gossipsub v1.1 Protocol Design + Implementation for Protocol Labs*. `https://leastauthority.com/blog/audit-of-gossipsub-v1-1-protocol-design-implementation-for-protocol-labs/`. Accessed 3 March 2022.

[37] Panagiotis Manolios (2000): *Correctness of Pipelined Machines*. In: *Formal Methods in Computer-Aided Design, FMCAD*, doi:10.1007/3-540-40922-X_11.

[38] Panagiotis Manolios (2001): *Mechanical Verification of Reactive Systems*. Ph.D. thesis, The University of Texas at Austin, Department of Computer Sciences, Austin TX.

[39] Panagiotis Manolios (2003): *A Compositional Theory of Refinement for Branching Time*. In: *Correct Hardware Design and Verification Methods, CHARME*, doi:10.1007/978-3-540-39724-3_28.

[40] Panagiotis Manolios, Kedar S. Namjoshi & Robert Summers (1999): *Linking Theorem Proving and Model-Checking with Well-Founded Bisimulation*. In: *Computer Aided Verification, CAV*, doi:10.1007/3-540-48683-6_32.

[41] Panagiotis Manolios & Sudarshan K. Srinivasan (2004): *Automatic Verification of Safety and Liveness for XScale-Like Processor Models Using WEB Refinements*. In: *Design, Automation and Test in Europe Conference and Exposition, DATE*, doi:10.1109/DATE.2004.1268844.

[42] Panagiotis Manolios & Sudarshan K. Srinivasan (2005): *Refinement Maps for Efficient Verification of Processor Models*. In: *Design, Automation and Test in Europe Conference and Exposition, DATE*, doi:10.1109/DATE.2005.257.

[43] Panagiotis Manolios & Sudarshan K. Srinivasan (2008): *Automatic verification of safety and liveness for pipelined machines using WEB refinement*. *ACM Trans. Design Autom. Electr. Syst.*, doi:10.1145/1367045.1367054.

[44] Panagiotis Manolios & Sudarshan K. Srinivasan (2008): *A Refinement-Based Compositional Reasoning Framework for Pipelined Machine Verification*. *IEEE Trans. Very Large Scale Integr. Syst.*, doi:10.1109/TVLSI.2008.918120.

[45] Panagiotis Manolios & Daron Vroon (2003): *Algorithms for Ordinal Arithmetic*. In: *Conference on Automated Deduction CADE*, doi:10.1007/978-3-540-45085-6_19.

[46] Panagiotis Manolios & Daron Vroon (2004): *Integrating Reasoning about Ordinal Arithmetic into ACL2*. In: *Formal Methods in Computer-Aided Design FMCAD*, LNCS, Springer–Verlag, doi:10.1007/978-3-540-30494-4_7.

[47] Panagiotis Manolios & Daron Vroon (2005): *Ordinal Arithmetic: Algorithms and Mechanization*. *Journal of Automated Reasoning*, doi:10.1007/s10817-005-9023-9.

[48] Panagiotis Manolios & Daron Vroon (2006): *Termination Analysis with Calling Context Graphs*. In: *Computer Aided Verification CAV*, doi:10.1007/11817963_36.

[49] Kenneth L. McMillan & Lenore D. Zuck (2019): *Formal specification and testing of QUIC*. In: *ACM Special Interest Group on Data Communication (SIGCOMM)*, doi:10.1145/3341302.3342087.

[50] Simon Meier, Benedikt Schmidt, Cas Cremers & David Basin (2013): *The TAMARIN prover for the symbolic analysis of security protocols*. In: *International conference on computer aided verification*, doi:10.1007/978-3-642-39799-8_48.

[51] Oded Padon, Giuliano Losa, Mooly Sagiv & Sharon Shoham (2017): *Paxos made EPR: decidable reasoning about distributed protocols*. *Proceedings of the ACM on Programming Languages*, doi:10.1145/3140568.

[52] Oded Padon, Kenneth L McMillan, Aurojit Panda, Mooly Sagiv & Sharon Shoham (2016): *Ivy: safety verification by interactive generalization*. In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, doi:10.1145/2908080.2908118.

[53] Fred B. Schneider (1990): *Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial*. *ACM Comput. Surv.*, doi:10.1145/98163.98167.

[54] William Schultz, Ian Dardik & Stavros Tripakis (2022): *Formal verification of a distributed dynamic reconfiguration protocol*. In: *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, doi:10.1145/3497775.3503688.

[55] Dimitris Vyzovitis: *gossipsub: An extensible baseline pubsub protocol*. `https://github.com/libp2p/specs/blob/master/pubsub/gossipsub/README.md`. Accessed 28 Nov 2022.

[56] Dimitris Vyzovitis (2020): *GossipSub v1.0: An extensible baseline pubsub protocol*. `https://github.com/libp2p/specs/blob/master/pubsub/gossipsub/gossipsub-v1.0.md`. Accessed 17 May 2022.

[57] Dimitris Vyzovitis (2020): *GossipSub v1.1: Security extensions to improve on attack resilience and bootstrapping*. `https://github.com/libp2p/specs/blob/master/pubsub/gossipsub/gossipsub-v1.1.md`. Accessed 3 March 2021.

[58] Dimitris Vyzovitis, Yusef Napora, Dirk McCormick, David Dias & Yiannis Psaras (2020): *GossipSub: Attack-resilient message propagation in the Filecoin and ETH2. 0 networks*. *arXiv preprint arXiv:2007.02754*.

[59] Dimitris Vyzovitis, Yusef Napora, Dirk McCormick, David Dias & Yiannis Psaras (2020): *Gossipsub-v1.1 Evaluation Report*. `https://gateway.ipfs.io/ipfs/QmRAFP5DBnvNjdYSbWhEhVRJJDFCLpPyvew5GwCCB4VxM4`. Accessed 21 May 2022.

[60] James R Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D Ernst & Thomas Anderson (2015): *Verdi: a framework for implementing and formally verifying distributed systems*. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, doi:10.1145/2737924.2737958.

[61] Doug Woos, James R Wilcox, Steve Anton, Zachary Tatlock, Michael D Ernst & Thomas Anderson (2016): *Planning for change in a formal verification of the Raft consensus protocol*. In: *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, doi:10.1145/2854065.2854081.

[62] Jianliang Wu, Ruoyu Wu, Dongyan Xu, Dave Jing Tian & Antonio Bianchi (2022): *Formal Model-Driven Discovery of Bluetooth Protocol Design Vulnerabilities*. doi:10.1109/SP46214.2022.9833777.

[63] Yuan Yu, Panagiotis Manolios & Leslie Lamport (1999): *Model checking TLA+ specifications*. In: *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, doi:10.1007/3-540-48153-2_6.

Proving Calculational Proofs Correct

Andrew T. Walter Ankit Kumar Panagiotis Manolios

Khoury College
Northeastern University
Massachusetts, USA

walter.a@northeastern.edu kumar.anki@northeastern.edu p.manolios@northeastern.edu

Teaching proofs is a crucial component of any undergraduate-level program that covers formal reasoning. We have developed a calculational reasoning format and refined it over several years of teaching a freshman-level course, "Logic and Computation", to thousands of undergraduate students. In our companion paper [28], we presented our calculational proof format, gave an overview of the calculational proof checker (CPC) tool that we developed to help users write and validate proofs, described some of the technical and implementation details of CPC and provided several publicly available proofs written using our format. In this paper, we dive deeper into the implementation details of CPC, highlighting how proof validation works, which helps us argue that our proof checking process is sound.

1 Introduction

Calculational Proof Checker (CPC) is a tool designed to help teach undergraduate computer science students how to write proofs. In a previous work [28] we presented the calculational proof format used by CPC and gave an overview of its design and implementation. Here we provide additional details about CPC's implementation and provide an argument for CPC's soundness.

CPC was designed for Manolios' freshman-level CS2800 "Logic and Computation" course [18], which uses the ACL2 Sedan (ACL2s) theorem prover [4, 12] to introduce logic and formal reasoning. ACL2s extends ACL2 with several additional features, including the `defdata` data definition framework [8], the `cgen` counterexample generation framework [5–7, 9], a termination analysis system using calling context graphs [23] and ordinals [20–22] and property-based modeling and analysis. As nearly anyone who has taught a formal reasoning class can attest to, teaching students how to identify what is a proof and what is not is challenging, and teaching students how to write proofs is even more so. The choice of proof format is highly impactful from a pedagogical standpoint, and therefore we put substantial effort into developing ours based on many years of experience teaching CS2800. The proof format we use in CS2800 is heavily inspired by the calculational proof style popularized by Dijkstra [10, 11]. Dijkstra's proof format is appropriate here because (1) its linear proofs are easier to check in a local manner (2) explicit context forces students to identify which parts of the context are used to discharge each step and (3) it is designed for human consumption rather than for a proof assistant, making these skills highly transferable.

CPC checks proofs in three *phases*. Phases 0 and 1 are intended to find problems with the proof in a way that is aimed at generating actionable and high-quality feedback for the user, and were discussed in detail in the companion paper [28]. Phase 2 involves translating the proof into one or more ACL2s theorems with proof-builder [1, `proof-builder`] instructions and checking these theorems inside of ACL2s. Therefore, the soundness of CPC reduces to the soundness of the proof-builder and ACL2s.

Our contributions include: (1) a method for translating calculational proofs into ACL2s theorems checkable by an unmodified ACL2s instance, (2) a proof of soundness of CPC and (3) several exten-

A. Coglio, S. Swords (Eds.): 18th ACL2 Workshop (ACL2-2023)
EPTCS 393, 2023, pp. 133–150, doi:10.4204/EPTCS.393.11

sions and libraries for ACL2s we developed for CPC. The source code for CPC is available in a public repository [27].

2 Proof Format

We illustrate our proof format with an example proof of a conjecture, shown in Figure 1. Notice that the proof document starts with ACL2s definitions of relevant functions required for the proof. We use ACL2s' `definec` to define functions with input and output types. We will discuss `definec` in more detail later, but for now one should read the definition `(definec aapp (a :tl b :tl) :tl ...)` as the definition of a function `aapp` that takes as argument two true lists `a` and `b` and returns a true list. We also use ACL2s' `property` form to state an ACL2 theorem. The first argument to that form describes type constraints on free variables used in the form; in this case, one can read `(property assoc-append (x :tl y :tl z :tl) <body>)` as `(defthm assoc-append (implies (and (tlp x) (tlp y) (tlp z)) <body>)`.

In our proof format, proofs and ACL2s expressions can be arbitrarily interleaved, allowing for example a user to define an ACL2s function, write a CPC proof about that function, and then use that proof to justify the admission of another ACL2s function. Users can also define helper lemmas in ACL2s using standard ACL2s proof techniques and subsequently apply those lemmas in a CPC proof, as is done in this example with the ACL2s lemma `assoc-append`.

The proof starts with a named conjecture "`revt-rrev-help`" followed by the expression to be proved. Note that it is not relevant to the proof checker whether one uses `Lemma` versus `Conjecture`, `Property` or `Theorem` to name a proof. In this case, we want to prove the conjecture using induction, so we specify that we are doing a proof by induction and provide the function that gives rise to the induction scheme we want to use. We then provide a number of subproofs (Induction Case 0 through 2), one for each proof obligation that induction using the specified scheme gives rise to. In this case, each of these is an equational reasoning proof, though in general any number of them could be induction proofs instead. For each equational reasoning proof, we provide an expression to be proved. If the expression is of the form $A \to B \to C$ then we require that the user use the logical rule of exportation to eliminate the nesting of implications and state $(A \wedge B) \to C$ in the *exportation step*. This must be done recursively, *e.g.* it should apply to an expression of that form in the antecedent of an implication. If desired, the statement may also be transformed into an equivalent one using propositional logic rules during this step. A contract completion step (which will be discussed in detail later) comes next, if applicable. The user then writes out the proof context (the hypotheses of the contract completed statement, if it is an implication) as a labeled list of *context items*. Additional context items can be added in the *derived context*. In Induction Case 2, derived context items are used to derive the consequent of the induction hypothesis so that it can be used more easily in the proof. Each derived context item has a list of justifications. If one can derive `nil` (false) in a derived context item, there is no need to provide the rest of the sections for that proof. The proof *goal* (the consequent of the contract completed statement if it is an implication, or the contract completed statement itself otherwise) is then listed, followed by a sequence of *proof steps*. Each proof step consists of two statements separated by a relation and a set of justifications for that step.

A simplified and compacted version of our proof grammar is shown in Figure 2. For brevity we do not include the complete grammar of our proof format, but it is available in our repository [27]. Our companion paper [28] has more examples of proofs written for CPC, both from CS2800 assignments and Dijkstra's EWDs [10]. We recommend that interested readers review those example proofs.

```
(definec aapp (a :tl b :tl) :tl
  (if (endp a)
      b
    (cons (first a) (aapp (rest a) b))))

(definec rrev (x :tl) :tl
  (if (endp x)
      nil
    (aapp (rrev (rest x)) (list (first x)))))

(definec revt (x :tl acc :tl) :tl
  (if (endp x)
      acc
    (revt (rest x) (cons (first x) acc))))

(property assoc-append (x :tl y :tl z :tl)
    (equal (aapp x (aapp y z))
           (aapp (aapp x y) z)))

Lemma revt-rrev-help:
(implies (and (tlp x)
              (tlp acc))
         (equal (revt x acc)
                (aapp (rrev x) acc)))

Proof by: Induction on (revt x acc)

Induction Case 0:
;; Elided case where (not (and (tlp x) (tlp acc)))
QED

Induction Case 1:
(implies (endp x)
  (implies (and (tlp x) (tlp acc))
           (equal (revt x acc)
                  (aapp (rrev x) acc))))

Exportation:
(implies (and (tlp x) (tlp acc) (endp x))
         (equal (revt x acc)
                (aapp (rrev x) acc)))

Context:
C1. (tlp x)
C2. (tlp acc)
C3. (endp x)

Derived Context:
D1. (equal x nil) { C1, C3 }

Goal: (equal (revt x acc) (aapp (rrev x) acc))

Proof:
(revt x acc)
== { D1, Def revt }
acc
== { Def aapp }
(aapp nil acc)
== { Def rrev, D1 }
(aapp (rrev x) acc)

QED
```

```
Induction Case 2:
(implies (and (not (endp x))
  (implies
    (and (tlp (cdr x))
         (tlp (cons (car x) acc)))
         (equal (revt (cdr x) (cons (car x) acc))
                (aapp (rrev (cdr x))
                      (cons (car x) acc)))))
  (implies (and (tlp x) (tlp acc))
           (equal (revt x acc)
                  (aapp (rrev x) acc))))

Exportation:
(implies
  (and (tlp x)
       (tlp acc)
       (not (endp x))
       (implies
         (and (tlp (cdr x))
              (tlp (cons (car x) acc)))
              (equal (revt (cdr x)
                           (cons (car x) acc))
                     (aapp (rrev (cdr x))
                           (cons (car x) acc)))))
  (equal (revt x acc)
         (aapp (rrev x) acc)))

Context:
C1. (tlp x)
C2. (tlp acc)
C3. (not (endp x))
C4. (implies (and (tlp (cdr x))
                  (tlp (cons (car x) acc)))
      (equal (revt (cdr x) (cons (car x) acc))
             (aapp (rrev (cdr x))
                   (cons (car x) acc))))

Derived Context:
D1. (tlp (cdr x)) { C1, C3, Def tlp }
D2. (tlp (cons (car x) acc)) { C2, C3, Def tlp }
D3. (equal (revt (cdr x) (cons (car x) acc))
         (aapp (rrev (cdr x)) (cons (car x) acc)))
    { D1, D2, C4, MP }

Goal: (equal (revt x acc) (aapp (rrev x) acc))

Proof:
(revt x acc)
== { Def revt, C3 }
(revt (cdr x) (cons (car x) acc))
== { D3 }
(aapp (rrev (cdr x)) (cons (car x) acc))
== { Def aapp, car-cdr axioms }
(aapp (rrev (cdr x)) (aapp (list (car x)) acc))
== { Lemma assoc-append ((x (rrev (cdr x)))
                (y (list (car x))) (z acc)) }
(aapp (aapp (rrev (cdr x)) (list (car x))) acc)
== { C3, Def rrev, car-cdr axioms }
(aapp (rrev x) acc)

QED

QED
```

Figure 1: An example proof written in our proof format. This proof file is available in our repo [27] at the path `example/ind-examples/pass/rrev.proof`.

⟨*ProofDocument*⟩ ::= (⟨*Proof*⟩ | ⟨*SExpression*⟩)+

⟨*Proof*⟩ ::= ⟨*Type*⟩ $\mathscr{V}$: $\mathscr{E}$ [Exportation: $\mathscr{E}$] [Contract Completion: $\mathscr{E}$] ⟨*Body*⟩ QED

⟨*Body*⟩ ::= ⟨*Simple*⟩ | ⟨*Inductive*⟩

⟨*Simple*⟩ ::= [Context: ⟨*Ctx*⟩] [Derived Context: ⟨*Dtx*⟩] Goal: $\mathscr{E}$ Proof: ⟨*Seq*⟩

⟨*Inductive*⟩ ::= Proof by: $\mathscr{E}$ [⟨*ContractCase*⟩] ⟨*BaseCase*⟩+ ⟨*InductionCase*⟩*

⟨*ContractCase*⟩ ::= Contract Case $\mathbb{N}$: $\mathscr{E}$ ⟨*Body*⟩ QED

⟨*BaseCase*⟩ ::= Base Case $\mathbb{N}$: $\mathscr{E}$ ⟨*Body*⟩ QED

⟨*InductionCase*⟩ ::= Induction Case $\mathbb{N}$: $\mathscr{E}$ ⟨*Body*⟩ QED

⟨*Type*⟩ ::= Conjecture | Property | Lemma | Theorem

⟨*Ctx*⟩ ::= (C$\mathbb{N}$: $\mathscr{E}$)*

⟨*Dtx*⟩ ::= (D$\mathbb{N}$: $\mathscr{E}$)*

⟨*Seq*⟩ ::= $\mathscr{B}$ ($\mathscr{R}$ {⟨*Hint*⟩ (, ⟨*Hint*⟩)*} $\mathscr{B}$)*

⟨*Hint*⟩ ::= ⟨*Type*⟩ $\mathscr{V}$ [$\mathscr{S}$] | C$\mathbb{N}$ | D$\mathbb{N}$ | Def $\mathscr{F}$ | $\mathscr{A}$ | algebra | obvious | PL | MP

Figure 2: EBNF grammar for our calculational proofs where, in the ACL2s universe, $\mathscr{V}$ is a fresh variable or natural number, $\mathscr{E}$ is an expression, $\mathbb{N}$ is a natural number, $\mathscr{B}$ is a Boolean expression, $\mathscr{R}$ is a binary relation on Boolean expressions, $\mathscr{S}$ is an association list used to represent a valid substitution, $\mathscr{F}$ is a valid function name, $\mathscr{A}$ is an axiom, and ⟨*SExpression*⟩ is an ACL2s event form. PL and MP stand for Propositional Logic and Modus Ponens hints respectively. Items in square brackets are optional.

3 System Architecture

We summarize the architecture of CPC here. We refer interested readers to our companion paper [28] for a detailed description. The architecture of CPC consists of three primary pieces — the user interface, the Xtext [31] language support, and the ACL2s backend. Xtext is a framework for building domain-specific languages, and automatically generates a lexer and parser from our proof format grammar. A user submits a proof document for checking through one of the CPC interfaces, which is parsed by Xtext and turned into an Xtext document. Next, an Xtext *validator* that we developed runs on the Xtext document, translating it into a form that is usable by the ACL2s backend and invoking that backend. The ACL2s backend then runs through the proof document and reports any issues back to the Xtext validator, which sends that information back to the user interface for reporting to the user. A major benefit of using Xtext in this way is the ability to associate errors detected by the backend with regions of the user's proof document. This means that, for example if a step is determined to be incorrect, we can produce error underlining for that step to help the user localize the error. Xtext also allows us to provide IDE features like syntax highlighting and code folding with minimal additional effort.

The ACL2s backend of CPC is implemented using our ACL2s Systems Programming methodology, which we described in an ACL2 Workshop paper last year [29]. That is, the backend is implemented mainly in "raw Lisp" and makes queries to ACL2s using the API described in our paper. This allows us to use programming constructs that are not legal in logic-mode ACL2s code, like the Common Lisp condition system.

As the proof format's grammar (see Figure 2) specifies, a proof document consists of a sequence of elements, where each element is either a proof or an event form. Here, an event form is a call to any of the ACL2s event functions, which are a superset of the ACL2 event functions [1, `events`]. In our examples, the most commonly used event forms consist of `property`, `defdata` and `definec`. When CPC is run on a proof document, it processes each element of the proof document in sequence, evaluating the element in ACL2s using `ld` [1, `ld`] if the element is an event form, or performing proof checking and generation if the element is a proof. Operating in this way adds some complexity but also makes CPC more flexible. For example, a user can define an ACL2s function, write a proof about that function, and then use that proof to justify the admission of another ACL2s function. This would not be possible if we only supported documents where a set of ACL2s expressions (or a book) ran before all of the proofs. If an event form evaluation or a proof check fails, CPC will report an error to the user but will continue to operate on subsequent elements inside the proof document.

As previously stated, CPC performs proof checking in three phases. Phases 0 and 1 are primarily designed to find problems in a proof document that we can provide actionable and high-quality feedback for. In Phase 2, CPC will translate the proof file into appropriate ACL2s theorems complete with proof-builder commands, which are then run through ACL2s to confirm that the proofs in that file are correct. Our soundness argument is based entirely on Phase 2.

4 Proof Checking

Our prior work [28] describes Phases 0 and 1 in detail. Here we will summarize Phases 0 and 1 and describe some relevant aspects in more detail.

Phase 0 is a quick, syntactic check of the proof document performed by Xtext. This is provided as part of the parser that Xtext generates from our grammar. Phase 1 is performed in the ACL2s backend, and can itself be broken up into performing three checks. The first is to check that the initial setup of the

proof—the contract completion, exportation, context and derived context, all of which we will discuss in more detail later—is correct. The second is to check that all of the steps (for a non-inductive proof) or all of the subproofs (for an inductive proof) are correct. For a non-inductive proof, the third step is to confirm that the conjunction of the steps is sufficient to prove the statement under consideration. For an inductive proof, the third step is to confirm that the subproofs constitute the proof obligations of the induction proof that the user specified.

4.1 Guards and Contract Completion

Students in CS2800 are taught to reason about programs. Using ACL2s' `defdata` [1, `defdata`] is helpful as it is a natural way to introduce students to contract-driven development. Students write all of their functions using ACL2s' `definec`, which requires that the user specify the type of input arguments to the function as well as the type that the function outputs. `definec` [1, `acl2s::definec`] is hooked into ACL2's guard system [1, `guard`] in such a way that a function defined using `definec` will have guards that assert that its arguments satisfy their specified types. Recall that the *guard obligations* for an expression is the sequence of conditions that must be true to satisfy the guards for every function call inside that expression. The *function contract* for a `definec` function is the statement that if all arguments to the function in some function call satisfy the specified input types (the input contract for the function is satisfied), that call evaluates to a value that satisfies the specified output type. When a `definec` form is evaluated, in addition to proving termination like `defun`, ACL2s will prove that the function's contract holds and will perform guard verification of the function's body. Guard verification is the process of proving that the guard obligations of an expression hold. Note that each `defdata` type has a "type predicate" associated with it—a function of one argument that evaluates to true if and only if that argument is a member of the corresponding type. The function contract theorems that `definec` submits are suffixed with `-CONTRACT` and `-CONTRACT-TP`.

We require that the statement a user is trying to prove in CPC has an empty sequence of guard obligations (equivalently, the guard obligations are all satisfied), or if this is not the case, we require that the user perform *contract completion* on the statement before proving it. Contract completion refers to the process of adding appropriate hypotheses to a statement to satisfy its guard obligations. We will also refer to the resulting statement after contract completion as the contract completion of the original statement. Performing contract completion on a statement of course changes the logical meaning of the statement. From a pedagogical standpoint, forcing users to perform contract completion helps us highlight the correspondence between the statements being proved and the code (the executable bodies of the functions in the statement). The way that `definec` works is relevant here—the logical definition of a function admitted using `definec` states that a call to the function with inputs that don't satisfy the function's input contract will evaluate to an arbitrary value satisfying the function's specified return type. A consequence of this is that a `definec` function cannot be expanded into its user-provided definition unless it is known that the function's input contract is satisfied. It is important to note that this is different from simply adding guards to a `defun`, as guards do not affect either the semantics of a function definition or the theorem prover [1, `guard-miscellany`]. Enforcing that statements are contract completed eliminates the possibility of errors or counterexamples due to guard violations ("type errors").

Note that the order in which hypotheses appear in a conjunction matters, as `and` in ACL2 is logically just syntactic sugar for `if` statements and the type information that a hypothesis provides might be necessary to satisfy the guards of a subsequent hypothesis. For example, if the original expression was `(implies (in e l) (consp l))` and the ACL2s definition of `in` requires that `(tlp l)`, the correct contract completion of the statement is `(implies (and (tlp l) (in e l)) (consp l))` and **not**

`(implies (and (in e l) (tlp l)) (consp l))`.

It would be simpler to enforce that users provide contract completed statements at the get-go, but having users perform contract completion inside of CPC has some advantages. In particular, having both the original statement and the contract completed statement inside of CPC allows us to check that the contract completion was done appropriately, *e.g.* that only the necessary hypotheses were added to satisfy the guard obligations. We do not guarantee CPC's soundness when the user provides a *non-trivial contract completion* — one that is not syntactically equivalent to the exported statement (if provided) or original statement (otherwise). To be clear, the only situation in which a non-trivial contract completion must be provided is when the original statement has a non-empty sequence of guard obligations, that is, there is at least one function call in the statement with an input contract that is not provably always true. If a user provides a non-trivial contract completion, we currently produce a warning notifying the user of the potential for unsoundness and recommending they update the original conjecture so that it is contract completed.

An example of a statement with a trivial contract completion is `(implies (and (tlp x) (tlp y)) (equal (app x y) (app y x)))`. Since the ACL2s definition of `app` only requires that its arguments are true lists, the two antecedents ensure that the arguments to `app` are true lists and both `tlp` and `equal` have no guards, the guard obligations for this statement are trivially true and thus no antecedents need to be added during contract completion.

An example of a statement with a non-trivial contract completion is `(implies (in e l) (consp l))`, given the definition of `in` requires that its second argument is a true list. Since the guard obligations for this statement (just `(tlp l)`) are not trivially true, a non-trivial contract completion is required. In this case, `(tlp l)` must be added as an antecedent before the `(in e l)` hypothesis, so the only possible correct contract completion is `(implies (and (tlp l) (in e l)) (consp l))`.

4.2 Proof Building Blocks

The basic building block of an equational reasoning proof is a proof step — a statement that two expressions satisfy some relation, justified by one or more hints. In general, a step in an equational reasoning proof in our format will look like (with α and β being S-expressions and R being either a relation or an alias for a relation):

```
α
R { H1, ..., Hn }
β
```

We say that this step is correct if and only if ACL2s can prove the statement `(R α β)` under an appropriate set of hypotheses and when constrained to an appropriate theory. As we will discuss shortly, the appropriate set of hypotheses and the appropriate theory both are influenced by the hints $H = \{H_1, ..., H_n\}$ that the user provided, but also by the context of the proof that the step is contained inside.

Hints

CPC supports several types of hints for justifying reasoning steps. These include `C`i and `D`i which refer to context and derived context items respectively, `def` *foo* which allows one to reference the definition of a function (allowing one to expand a function call into its body with an appropriate substitution) and `arithmetic` which allows many kinds of arithmetic manipulations. Some hints have aliases (for example, `arith` is an alias for `arithmetic`). Other hints only exist for readability—for example, we use `MP` (`Modus Ponens`) to indicate that a step or derived context item is justified by the conclusion of an

implication after satisfying that implication's hypotheses, but it does not affect CPC's checking. Each hint for a proof step gives rise to zero or more of hypotheses (*Hyps*), *Rules* and *Lemma instantiations*, used in proving the proof step. *Rules* here refer to proved theorems in ACL2's database, which ACL2 can automatically apply. We define functions $\text{hyps}(h)$, $\text{rules}(h)$ and $\text{instances}(h)$ to be the set of hypotheses, rules and lemma instantiations (in a format amenable to ACL2) that a hint h gives rise to, respectively.

- `C`*i*: add the expression corresponding to the i^{th} context item as a hypothesis
- `D`*i*: add the expression corresponding to the i^{th} derived context item as a hypothesis
- `def` *foo*: enable the definition rule(s) for the function *foo*
- `cons axioms`: enable the following rules regarding `cons`: `(:rewrite car-cons)`, `(:rewrite cdr-cons)`, `car-cdr-elim`, `cons-equal`, `default-car`, `default-cdr`, `cons-car-cdr`
- `arithmetic`: enable the set of rules added by including the `arith-5` books
- `evaluation`: enable all rules of type `:executable-counterpart`
- `lemma` *foo*: add a lemma instance `:use` hint for *foo* with the given instantiation (if provided). This effectively instantiates the given lemma and adds the resulting expression as a hypothesis.

Theories

At different times during both Phases 1 and 2, it is useful to be able to ask ACL2s to prove a statement while limiting the types of reasoning that it can use. One of the ways we do this is by controlling the set of rules that ACL2s has access to. We define theories for certain sets of rules that are used inside CPC:

- `arith-5-theory` is the set of rules that are added by including the `"arithmetic-5/top"` book in a vanilla ACL2 instance.
- `min-theory` consists of ACL2's minimal theory (which includes only rules about basic built-in functions like `if` and `cons`) plus `(:executable-counterpart acl2::tau-system)`, `(:compound-recognizer booleanp-compound-recognizer)`, and `(:definition not)`. The former of these three rules enables ACL2 to perform some type-based reasoning, and the latter two are often useful for reasoning about propositional logic.
- `arith-theory` which consists of some basic facts about `+` and $*$.
- `type-prescription-theory` which consists of any rules of type `:type-prescription`
- `executable-theory` which consists of any rules of type `:executable-counterpart`.
- `contract-theory` which is `min-theory` plus `type-prescription-theory` and any rules with names ending in `"CONTRACT"` or `"CONTRACT-TP"`. The latter rules correspond to the function contracts for any functions admitted using `definec`.
- `min-executable-theory` which is the union of the rules in `min` and `executable`.

Type Hypotheses

Almost any proof involving a function defined with `definec` requires that the function's input contract is satisfied. In early versions of CPC, we found that this resulted in users needing to repeatedly include justifications in their steps corresponding to hypotheses that some free variables in the

proof statement satisfy some type predicates. Given that users already must perform contract completion on their proof statement, this felt like an unnecessary burden. Therefore for any step or derived context item, CPC will automatically include hints that correspond to calls of type predicates. For example, in `Induction Case 2` in the proof example in Section 2, `C1. (tlp x)`, `C2. (tlp acc)`, `D1. (tlp (cdr x))` and `D2. (tlp (cons (car x) acc))` are all included "for free" as justifications of any proof step.

5 Soundness

Once the user has provided a proof that passes Phases 0 and 1, we would like to translate it into an ACL2s theorem. There are two benefits this brings: (1) we can reduce the soundness of CPC to that of ACL2s and (2) it enables one to perform a proof in CPC that might be challenging to do in ACL2s and then use the resulting theorem in ACL2s. The second benefit is not currently exposed in a convenient way to users of CPC, but we believe it would be easy to implement this feature.

It is important to note that ACL2s contains extensions to ACL2 that require trust tags [1, `defttag`] and perform potentially unsafe modifications to ACL2. Therefore, we can only reduce the soundness of CPC to the soundness of ACL2s, not further to the soundness of ACL2.

Our soundness theorem is as follows: given a proof P without a non-trivial contract completion and whose proof statement is ϕ, if CPC validates P then ϕ is a valid statement in ACL2s, given the same ACL2s world prior to the validation of P. The witness for our soundness theorem is the ACL2s theorem that proves ϕ.

This theorem is easy to prove, as CPC will validate a proof only if it was able to prove that proof's statement in ACL2s, using the proof-builder instructions that CPC generates as described below. Note that we make no claims about completeness—CPC may reject a proof of a valid statement.

5.1 Proof Builder

Generating an ACL2s statement of a CPC theorem is straightforward, but we do not want to simply hand this statement off to ACL2s for an automatic proof—ACL2s may decide to attempt to take a different proof approach that requires a different set of lemmas, or may just fail to find a proof. Ultimately our goal is to determine whether or not the user's proof is correct, so we should be able to transform it and its justifications into a theorem that ACL2s can prove. For this reason, we use ACL2's proof-builder functionality, which allows us to command the theorem prover's behavior at a much lower level.

The proof-builder operates in a manner similar to an interactive proof assistant like Coq [3] or Isabelle [24]: there is a *proof state* consisting of a stack of goals, each of which contains a set of hypotheses and a statement to be proved, and one provides *instructions* that operate on the goal stack. These instructions range in granularity, with coarse instructions like `prove` (attempt to prove the current goal entirely automatically with ACL2's full power) to fine instructions like `dive` (focus on a particular subexpression in the current statement to be proved). ACL2's documentation provides information about many of the available proof-builder instructions [1, `proof-builder-commands`]. For CPC we developed several new proof-builder instructions, many of which are variants of existing instructions that succeed where the existing instructions would fail. For example, `:retain-or-skip` [1, `acl2-pc::retain-or-skip`] is exactly like the built-in `:retain` [1, `acl2-pc::retain`] instruction, except that it will succeed even when all of the existing hypotheses are retained (producing no change in the proof-builder state). Many instructions have similar behavior that is desirable when a human is interacting directly with the proof-

builder, but that is not when automatically generating instructions. These new proof-builder instructions are available in the ACL2 distribution, inside `books/acl2s/utilities.lisp`. All of the new instructions are listed below:

- `:claim-simple`: exactly like `:claim`, except that it does not automatically perform hypothesis promotion on the newly created goal.
- `:pro-or-skip`: exactly like `:pro`, except that it will succeed even when no promotion is possible.
- `:drop-or-skip`: exactly like `:drop`, except that it will succeed even when there are no top-level hypotheses and no arguments are provided.
- `:retain-or-skip`: exactly like `:retain`, except that it will succeed even when all of the existing hypotheses are retained.
- `:cg-or-skip`: exactly like `:cg`, except that it will succeed even when the specified goal to change to is the same as the current goal.
- `:instantiate`: instantiate a theorem as a hypothesis under the given substitution.
- `:split-in-theory`: exactly like `:split`, except that a theory can be provided to use instead of `minimal-theory`.
- `:by`: prove a goal using exactly an existing lemma under a given substitution.

5.2 Instruction Generation Algorithms

We will now describe how we generate proof-builder instructions for steps and derived context items, equational reasoning proofs, and inductive proofs. In the below algorithm listings, we will use a typewriter font face `like this` to denote S-expressions that we generate. Some additional comments on notation:

- $x +\!\!+ y$ denotes the sequence produced by appending the sequences x and y. If y is a set, then it is first transformed into a sequence by enumerating the elements of y in an arbitrary order.
- An ACL2s statement x is a *type predicate call* if and only if it is a function call with one argument and the function name is known by `defdata` to be a type predicate.
- Let $\text{hid}(x)$ be an identifier used by the proof-builder to refer to the hypothesis corresponding to the context or derived context item x.
- Let $\text{rules}(x)$ be the set of rules that a hint x gives rise to.
- Let $\text{instances}(x)$ be the set of lemma instantiations (in a format amenable to ACL2) that a hint x gives rise to.
- `IndObsAndNames(`*stmt, indterm*`)` calls ACL2's proof-builder to determine what goals are created when one tries to prove *stmt* by performing an induction on *indterm*. The output is a set of tuples $(obs, name)$ where *obs* is an ACL2s statement expressing one of the created goals and *name* is the name that ACL2s gave to that goal.

In the below algorithms, we elide the complexity of matching up the names that the proof-builder gives to the hypotheses with the names of context items that the user gave in the proof.

```
;; A step
α
R { H1, ..., Hn }
β
;; A derived context item
Dn. γ { H1, ..., Hn }
```

Figure 3: The general form of a proof step and a derived context item.

Hints to Instructions

The processes of generating proof-builder instructions for a step and for a derived context item are similar, so we will describe them together. Figure 3 shows the general form of a step and a derived context item. Using names from Figure 3, we will define the *equivalent expression* of a step to be `(R α β)` and the equivalent expression of a derived context item to be γ. Algorithm 1 is the corresponding algorithm for this process. We pass the equivalent expression of the step or derived context item into the `stmt` input of the algorithm. Let EE be the equivalent expression of the step or derived context item in question.

The instructions that we generate should do the following: use `:claim-simple` to add a hypothesis that the equivalent expression of the step or derived context item holds, then prove that this hypothesis holds using the justifications that the user provided.

We first generate a `claim-simple` instruction with EE as the statement to cause the proof-builder to add EE as a hypothesis in the current goal. This also results in the creation of a new goal to prove EE given the current set of hypotheses (before EE was added). We pass `:hints :none` to the `claim-simple` instruction so that ACL2s does not try to prove this new goal automatically. Then, we generate a `cg` instruction to change to the newly generated goal. Next, we calculate the guard obligations that this goal would have given the current context and generate a `claim` instruction with those obligations as its statement. This `claim` instruction will result in ACL2s trying to prove that the statement holds automatically. Next, based on $H_1, ..., H_n$, we determine which context and derived context items should be available when proving EE. We then generate a `retain-or-skip` instruction with appropriate arguments to only retain the appropriate context and derived context items. We then determine based on $H_1, ..., H_n$ what ACL2 rules should be available. We generate a `in-theory` instruction with appropriate arguments to enable and disable rules appropriately. Finally, we generate the instruction `(:finish :bash)`, which tells ACL2s to attempt to prove the current goal while limiting its abilities. If ACL2s is unable to prove the goal, it will raise an error and the proof attempt will result in a failure.

Equational reasoning proofs

Generating instructions for an equational reasoning proof is fairly straightforward; the algorithm is shown in Algorithm 2.

We start with `:pro-or-skip` to expand the proof statement's implication into antecedents and a consequent (if it is an implication). Then, we generate instructions to add a hypothesis for each derived context item and prove that it holds given the provided justifications. We do something very similar for each proof step. Then, we `:demote` to turn the goal and hypotheses into an ACL2 implication statement before repeatedly calling `(:split-in-theory min-executable-theory)` until the goal has been discharged. We use `:split-in-theory` (and therefore `:split`) here as it is a convenient way to invoke ACL2 with very limited reasoning ability (just simplification, preprocessing, and whatever rules are in the given theory).

Algorithm 1: proof-builder instruction generation for a step or derived context item

Function `ProveUsingHints(`*stmt, hints, ctx*`)`

Input: *stmt* is the statement to prove, *hints* is the set of hints the user provided, and *ctx* is the set of context and derived context items it should be proved under.

$I \leftarrow []$;

`/* Add` *stmt* `as a hypothesis and as a new goal, do not attempt to prove it automatically, and switch to the new goal */`

$I \leftarrow I +\!\!+ [$`(:claim-simple` *stmt*`... :hints :none),:cg`$]$;

$hyps \leftarrow \{x \mid x \in hints \wedge x$ is a context or derived context hint $\}$;

$contracts \leftarrow \mathrm{GO}((\bigwedge_{h \in hyps} h) \rightarrow stmt)$;

if $contracts \neq$ `true` **then**

`/* Add` *contracts* `as a hypothesis and as a new goal & prove it automatically. */`

$I \leftarrow I +\!\!+ [$`(:claim` *contracts*`...)`$]$;

$typectx \leftarrow \{x \mid x \in ctx \wedge x$ is a type-predicate call $\}$;

$contractsidx \leftarrow$ a set containing the identifier of the *contracts* hypothesis if $contracts \neq$ `true` or $\emptyset$ otherwise;

`/* Only keep the hypotheses we should have given the hints the user provided */`

$I \leftarrow I +\!\!+ [$`(:retain-or-skip` $\{\mathrm{hid}(x) \mid x \in hyps \cup typectx \cup contractsidx\}$`)`$]$;

$hintrules \leftarrow \bigcup \{\mathrm{rules}(x) \mid x \in hints\}$;

`/* Ensure that only the rules that we should have access to given the user's hints are available */`

$I \leftarrow I +\!\!+ [$`(:in-theory (union-theories (theory 'contract-theory)` *hintrules*`))`$]$;

$lemmainstances \leftarrow \bigcup \{\mathrm{instances}(x) \mid x$ is a hint for $Dx_i\}$;

`/* Add any lemma instances that the user described in the hints */`

$I \leftarrow I +\!\!+ \{$`(:instantiate` x`)` $\mid x \in lemmainstances\}$;

`/* Ask ACL2 to automatically prove this goal without induction, and then reset to the original theory */`

$I \leftarrow I +\!\!+ [$`(:finish :bash),:in-theory`$]$;

return I

Algorithm 2: proof-builder instruction generation for a non-inductive conjecture

```
1  Function EquationalReasoningTranslate(C, D, R, P, H)
       Input: C and D are the sets of all non-derived context and derived context items for a proof
              respectively. R, P and H are the relations, step statements and hints for the proof's
              proof steps, indexed from the start of the proof. This function only operates on
              equational reasoning proofs.
2      I ← [];
       /* Perform exportation and expand implication into hyps/conclusion                    */
3      I ← I ++ [:pro-or-skip];
       /* Generate instructions for each derived context item                                */
4      foreach i ∈ [1..m] do
5          stmt ← the proof statement associated with Dx_i;
           /* Do not include any later derived context items in the context used to prove
              Dx_i                                                                           */
6          ctx ← C ∪ {Dx_j | j ∈ [1..i−1]};
7          I ← I ++ ProveUsingHints(stmt, hints, ctx);
       /* Generate instructions for each step                                                */
8      foreach i ∈ [1..n] do
9          stmt ← ( R_i P_i P_{i+1} );
10         ctx ← C ∪ D;
11         I ← I ++ ProveUsingHints(stmt, H_i, ctx);
       /* Turn the hypotheses and goal into an implication                                   */
12     I ← I ++ [:demote];
       /* Repeatedly call :split-in-theory until the proof is successful or we reach a
          fixpoint                                                                           */
13     I ← I ++
        [(:finish (:repeat-until-done (:split-in-theory min-executable-theory)))];
14     return I
```

Inductive proofs

The algorithm for generating proof-builder instructions for inductive proofs is provided in Algorithm 3.

Assume we have a proof by induction without a non-trivial contract completion. This can be thought of as several separate proofs, one for each induction proof obligation. For each of these subproofs, we will generate a separate ACL2s proof, complete with proof-builder instructions. Then, we generate an ACL2s proof for the top-level induction proof with instructions to perform a proof by induction using the induction scheme that the user specified, and generate instructions that discharge each proof obligation using the corresponding generated ACL2s proof. This approach requires that CPC determine the order of the subgoals that ACL2s will generate when asked to perform an induction proof with the given induction scheme so that we can map up the subproofs that the user performed with these subgoals, and thus in the instructions for each subgoal we can refer to the appropriate generated ACL2s proof. We generate these subgoals by using the ACL2 function `state-stack-from-instructions`, which allows one to get the state of the proof builder after running a sequence of instructions, and then reuse some existing CPC code to find a bijection between these subgoals and the induction proof cases that the user provided.

Once we have generated proof-builder instructions for an inductive proof, we generate `defthms` with proof-builder instructions for all of its subproofs. We then generate an `encapsulate` statement and insert all of the subproof `defthms` in the encapsulate as `local`. The inductive proof itself is not inserted into a `local` and is thus exported from the `encapsulate`. We do not want to export the subproof `defthms`, as they are only needed to show that the top-level inductive proof theorem holds.

6 Related Work

Our previous work [28] contains a longer discussion of works surrounding the use and mechanical verification of calculational proofs. Below we provide a summary of that discussion, as well as some related work in ACL2 in particular.

Calculational proofs were popularized by in the early 1990s by Dijkstra and Scholten [11], Gasteren [13] and Gries [14]. A series of works [2, 15, 16, 25] by Robinson, Stables, Back, Grundy and Wright resulted in the development of structured calculational proofs, an extension of the calculational proof style that allows for the hierarchical decomposition of proofs. This format reduces to natural deduction, but maintains the benefits of calculational proofs while also allowing for improved readability and browsability of proofs.

Manolios argued for the formalization of calculational proofs and their mechanized checking in 2000 [19]. Mizar [26] is a system for checking calculational proofs first developed in the 1970s. Several systems inspired by Mizar have been developed since, including Isabelle/Isar [30] and Leino et. al's poC extension to Dafny [17]. These systems typically follow Mizar's format in not requiring the user to explicitly state the proof context. Mizar has only lightweight support for automated reasoning in proving that proof steps hold and only allows equality relations inside of proofs. Isar allows for arbitrary relations and provides access to Isabelle's powerful reasoning capabilities, like *simp* for Isabelle's simplifier, and *auto* for a combination of several tools [17]. poC only allows a predefined set of relations but is as declarative as Mizar is, while providing more powerful automated reasoning with its SMT solver backend.

Algorithm 3: proof-builder instruction generation for an inductive conjecture

Function `InductiveTranslate`(*M, stmt, indterm, PC*)

Input: This function only operates on inductive proofs. M is a function mapping the names of the proof cases of this inductive proof to corresponding ACL2 theorems. *stmt* is the proof statement for this inductive proof, and *indterm* is the induction term. PC is the set of proof cases given for this inductive proof, where name(PC_i) is the name of PC_i and stmt(PC_i) is the proof statement for PC_i.

`/* Generate the proof obligations an induction on indterm will give rise to */`

obsnames ← `IndObsAndNames(`*stmt, indterm*`)`;

`/* Perform exportation, expand implication into hyps/conclusion, perform induction */`

$I \leftarrow$ [`:pro-or-skip`, `(:induct` *indterm*`)`];

Attempt to find an injective mapping from *obsnames* to PC, where an element $(obs, name) \in obsnames$ is mapped to an element $PC_i \in PC$ iff the conjunction of the hypotheses of *obs* after exportation are propositionally equivalent to the conjunction of the hypotheses of stmt(PC_i) after exportation.;

if *no such mapping exists* **then**

  raise an error;

inj ← the injective mapping;

foreach $(obs, name) \in obsnames$ **do**

  injPC ← $inj((obs, name))$;

  `/* Change to the induction obligation, use the existing proof to discharge it */`

  $I \leftarrow$ [`(:cg-or-skip` *name*`)`, `(:finish :demote (:by` M(name(*injPC*))`))`];

return I

7 Conclusion and Future work

We have presented an argument for the soundness of CPC, based on its translation of calculational proofs into ACL2s theorems with proof-builder instructions. We are interested in seeing how CPC can be used by "professional" users to design their proofs, and have some ideas about functionality that would be appropriate for these users. In particular, we see the need to provide more automation to such users, for example automatic generation of context and induction proof obligations or a "bash" mode for eliding simple subproofs like contract cases in inductive proofs. We hope to continue to extend and improve CPC based on requests from students and the community, and plan on continuing to use it to help teach undergraduates how to write proofs.

Acknowledgments

We sincerely thank Ken Baclawski, Raisa Bhuiyan, Harsh Chamarthi, Peter Dillinger, Robert Gold, Jason Hemann, Andrew Johnson, Alex Knauth, Michael Lin, Riccardo Pucella, Sanat Shajan, Olin Shivers, David Sprague, Atharva Shukla, Ravi Sundaram, Stavros Tripakis, Thomas Wahl, Josh Wallin, Kanming Xu and Michael Zappa for their help with developing and teaching with CPC. We also thank all of the Teaching Assistants and students in CS2800, who are too numerous to list individually, who used and provided feedback on CPC.

References

[1] ACL2 Contributors: *ACL2 XDoc Documentation*. `https://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/index.html`.

[2] Ralph Back, Jim Grundy & Joakim von Wright (1997): *Structured Calculational Proof*. *Formal Aspects of Computing* 9(5-6), pp. 469–483, doi:`10.1007/BF01211456`.

[3] Yves Bertot & Pierre Castéran (2004): *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series, Springer, doi:`10.1007/978-3-662-07964-5`.

[4] Harsh Chamarthi, Peter C. Dillinger, Panagiotis Manolios & Daron Vroon (2011): *The "ACL2" Sedan Theorem Proving System*. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, doi:`10.1007/978-3-642-19835-9_27`.

[5] Harsh Raju Chamarthi (2016): *Interactive Non-theorem Disproving*. Ph.D. thesis, Northeastern University, doi:`10.17760/D20467205`.

[6] Harsh Raju Chamarthi, Peter C. Dillinger, Matt Kaufmann & Panagiotis Manolios (2011): *Integrating Testing and Interactive Theorem Proving*. In David S. Hardin & Julien Schmaltz, editors: *Proceedings 10th International Workshop on the ACL2 Theorem Prover and its Applications*, *EPTCS* 70, pp. 4–19, doi:`10.4204/EPTCS.70.1`.

[7] Harsh Raju Chamarthi, Peter C. Dillinger, Matt Kaufmann & Panagiotis Manolios (2011): *Integrating Testing and Interactive Theorem Proving*. In: *International Workshop on the ACL2 Theorem Prover and its Applications*, EPTCS, doi:`10.4204/EPTCS.70.1`.

[8] Harsh Raju Chamarthi, Peter C. Dillinger & Panagiotis Manolios (2014): *Data Definitions in the ACL2 Sedan*. In: *Proceedings Twelfth International Workshop on the ACL2 Theorem Prover and its Applications*, EPTCS, doi:`10.4204/EPTCS.152.3`.

[9] Harsh Raju Chamarthi & Panagiotis Manolios (2011): *Automated specification analysis using an interactive theorem prover*. In Per Bjesse & Anna Slobodová, editors: *International Conference on Formal Methods*

in Computer-Aided Design, FMCAD '11, FMCAD Inc., pp. 46–53. Available at https://dl.acm.org/doi/10.5555/2157654.2157665.

[10] Edgar W. Dijkstra: *EWDs*. https://www.cs.utexas.edu/users/EWD/.

[11] Edsger W. Dijkstra & Carel S. Scholten (1990): *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science, Springer, doi:10.1007/978-1-4612-3228-5.

[12] Peter C. Dillinger, Panagiotis Manolios, Daron Vroon & J. Strother Moore (2007): *ACL2s: "The ACL2 Sedan"*. In: *Proceedings of the 7th Workshop on User Interfaces for Theorem Provers (UITP 2006)*, Electronic Notes in Theoretical Computer Science, doi:10.1016/j.entcs.2006.09.018.

[13] Antonetta J. M. van Gasteren (1990): *On the Shape of Mathematical Arguments*. *Lecture Notes in Computer Science* 445, Springer, doi:10.1007/BFb0020908.

[14] David Gries (1991): *Teaching Calculation and Discrimination: A More Effective Curriculum*. *Communications of the ACM* 34(3), pp. 44–55, doi:10.1145/102868.102870.

[15] Jim Grundy (1996): *A browsable format for proof presentation*. *Logic, Mathematics, and the Computer–Foundations: History, Philosophy and Applications* 14, pp. 171–178. Available at https://www.researchgate.net/publication/2359706_A_Browsable_Format_for_Proof_Presentation.

[16] Jim Grundy (1996): *Transformational Hierarchical Reasoning*. *The Computer Journal* 39(4), pp. 291–302, doi:10.1093/comjnl/39.4.291.

[17] K. Rustan M. Leino & Nadia Polikarpova (2013): *Verified Calculations*. In Ernie Cohen & Andrey Rybalchenko, editors: *Verified Software: Theories, Tools, Experiments - 5th International Conference, VSTTE 2013, Menlo Park, CA, USA, May 17-19, 2013, Revised Selected Papers*, *Lecture Notes in Computer Science* 8164, Springer, pp. 170–190, doi:10.1007/978-3-642-54108-7_9.

[18] Panagiotis Manolios: *Logic and Computation Class*. https://www.ccs.neu.edu/home/pete/courses/Logic-and-Computation/2022-Fall/.

[19] Panagiotis Manolios & J Strother Moore (2001): *On the desirability of mechanizing calculational proofs*. *Information Processing Letters* 77(2-4), pp. 173–179, doi:10.1016/S0020-0190(00)00200-3.

[20] Panagiotis Manolios & Daron Vroon (2003): *Algorithms for Ordinal Arithmetic*. In Franz Baader, editor: *19th International Conference on Automated Deduction (CADE)*, *Lecture Notes in Computer Science* 2741, Springer, pp. 243–257, doi:10.1007/978-3-540-45085-6_19.

[21] Panagiotis Manolios & Daron Vroon (2004): *Integrating Reasoning About Ordinal Arithmetic into ACL2*. In Alan J. Hu & Andrew K. Martin, editors: *5th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, *Lecture Notes in Computer Science* 3312, Springer, pp. 82–97, doi:10.1007/978-3-540-30494-4_7.

[22] Panagiotis Manolios & Daron Vroon (2005): *Ordinal Arithmetic: Algorithms and Mechanization*. *Journal of Automated Reasoning* 34(4), pp. 387–423, doi:10.1007/s10817-005-9023-9.

[23] Panagiotis Manolios & Daron Vroon (2006): *Termination Analysis with Calling Context Graphs*. In Thomas Ball & Robert B. Jones, editors: *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, *Lecture Notes in Computer Science* 4144, Springer, pp. 401–414, doi:10.1007/11817963_36.

[24] Tobias Nipkow, Lawrence C. Paulson & Markus Wenzel (2002): *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. *Lecture Notes in Computer Science* 2283, Springer, doi:10.1007/3-540-45949-9.

[25] Peter J. Robinson & John Staples (1993): *Formalizing a Hierarchical Structure of Practical Mathematical Reasoning*. *Journal of Logic and Computation* 3(1), pp. 47–61, doi:10.1093/logcom/3.1.47.

[26] Piotr Rudnicki (1992): *An overview of the Mizar project*. In: *Proceedings of the 1992 Workshop on Types for Proofs and Programs*.

[27] Andrew T. Walter, Ankit Kumar & Panagiotis Manolios: *Calculational Proof Checker repository*. https://gitlab.com/acl2s/proof-checking/calculational-proof-checker.

[28] Andrew T. Walter, Ankit Kumar & Panagiotis Manolios (2023): *Calculational Proofs in ACL2s*. arXiv:2307.12224.

[29] Andrew T. Walter & Panagiotis Manolios (2022): *ACL2s Systems Programming*. In: *Proceedings of the Seventeenth International Workshop on the ACL2 Theorem Prover and its Applications*, EPTCS, doi:10.4204/EPTCS.359.12.

[30] Makarius Wenzel (2007): *Isabelle/Isar–a generic framework for human-readable proof documents*. *Studies in Logic, Grammar and Rhetoric* 10(23). Available at http://mizar.org/trybulec65.

[31] Xtext Contributors: *Xtext*. Available at https://www.eclipse.org/Xtext/. Accessed on April 25th, 2022.

ACL2 Proofs of Nonlinear Inequalities with Imandra

Grant Passmore
Imandra Inc.
Austin, TX
Clare Hall, Cambridge
grant@imandra.ai

We present a proof-producing integration of ACL2 and Imandra for proving nonlinear inequalities. This leverages a new Imandra interface exposing its nonlinear decision procedures. The reasoning takes place over the reals, but the proofs produced are valid over the rationals and may be run in both ACL2 and ACL2(r). The ACL2 proofs Imandra constructs are extracted from Positivstellensatz refutations, a real algebraic analogue of the Nullstellensatz, and are found using convex optimization.

1 Introduction

Nonlinear inequalities can pose critical formal verification challenges. While nonlinear integer arithmetic is undecidable, nonlinear real arithmetic is decidable, and advances in decision procedures have brought many useful classes of problems within reach of automated methods. Unfortunately, most effective modern methods, e.g., those based on Cylindrical Algebraic Decomposition (CAD) [3], are not *proof producing* and rely on nontrivial computer algebra computations which must be trusted. This presents a major barrier for taking advantage of such techniques in formal proofs.

In this work, we use the Positivstellensatz [6, 15], a fundamental result in real algebraic geometry, to construct fully formal proofs of nonlinear real inequalities in ACL2. The Positivstellensatz guarantees the existence of proofs of inequalities in a certain formal system, and advances in convex optimization (including semidefinite programming (SDP) and sums-of-squares decompositions) allow us to effectively search over a convex space of *certificates* to find such proofs. When these proofs are found, we can then translate them into ACL2 proofs in a structured form that ACL2 can easily check.

Let us motivate our discussion with an example. Consider one direction of the discriminant criterion for solubility of a quadratic equation:

$$\forall x,a,b,c \in \mathbb{R}\left(ax^2+bx+c=0 \implies b^2-4ac \geq 0\right).$$

To prove this, we will negate and normalize its constraints s.t. all relations are drawn from $\{=,\geq,>,\neq\}$:

$$ax^2+bx+c=0 \wedge 4ac-b^2>0$$

and then proceed to derive a contradiction. The Positivstellensatz (cf. Sec 3) guarantees the existence of a *certificate* establishing unsatisfiability by a particularly simple form of argument. In this case, a certificate is given by

$$(4ac-b^2)+(2ax+b)^2+(-4a)(ax^2+bx+c)$$

as $(4ac-b^2)>0$ and $(-4a)(ax^2+bx+c)=0$ by assumption, and $(2ax+b)^2 \geq 0$ as it is a square. Thus, by assumption, the certificate must be strictly positive. But by polynomial arithmetic, it is easy to verify that the certificate sums to 0. Thus the negation of our conjecture implies that $0>0$. This is the general structure of a Positivstellensatz refutation, and all ACL2 proofs we produce proceed in this way.

A. Coglio, S. Swords (Eds.): 18th ACL2 Workshop (ACL2-2023)
EPTCS 393, 2023, pp. 151–160, doi:10.4204/EPTCS.393.12

2 The Imandra-ACL2 Interface and ACL2 Proofs

We have implemented the Positivstellensatz proof method in Imandra [11], and built an interface which extracts ACL2 proofs from its refutations.

To use it, one poses a conjecture as an S-expression in which all variables are implicitly taken to be reals. If Imandra is successful, an ACL2 proof is produced as an `(ENCAPSULATE ...)` event which exports a single theorem named `FINAL`. Note that in the ACL2 theorems we prove, the variables are `RATIONAL` instead of `REAL`, as `REALP` is available only in ACL2(r).

For our quadratic equation example, the input is:

```
(IMPLIES (= (+ (* A X X) (* B X) C) 0)
         (>= (- (* B B) (* 4 A C)) 0))
```

and the output (produced in a fraction of a second) is:

```
(ENCAPSULATE ()

;; Preamble

 (SET-IGNORE-OK T)
 (SET-IRRELEVANT-FORMALS-OK T)

 (LOCAL (DEFMACRO NEQ (X Y)
          `(OR (< ,X ,Y) (> ,X ,Y))))

 (LOCAL (DEFUN SQUARE (X)
          (* X X)))

 (LOCAL (DEFTHM SQUARE-PSD
          (IMPLIES (RATIONALP X)
                   (>= (SQUARE X) 0))
          :RULE-CLASSES (:LINEAR)))

 (LOCAL (DEFTHM SQUARE-TYPE
          (IMPLIES (RATIONALP X)
                   (RATIONALP (SQUARE X)))
          :RULE-CLASSES (:TYPE-PRESCRIPTION)))

 (LOCAL (IN-THEORY (DISABLE SQUARE)))

 (LOCAL (include-book "arithmetic-5/top" :dir :system))

;; Normalized problem polynomials

 (LOCAL (DEFUND PROB-0 (A B C X)
          (+ (* A (* X X)) (+ (* B X) C))))
```

```
 (LOCAL (DEFUND PROB-1 (A B C X)
          (- 0 (- (* B B) (* 4 (* A C))))))

;; Normalized goal expressed using problem polynomials

 (LOCAL (DEFUN GOAL (A B C X)
          (IMPLIES (AND (RATIONALP A)
                        (RATIONALP B)
                        (RATIONALP C) (RATIONALP X))
                   (NOT (AND (= (PROB-0 A B C X) 0)
                             (> (PROB-1 A B C X) 0)))))

;; Ideal cofactors

 (LOCAL (DEFUND IDEAL-CF-0 (A B C X)
          (* -4 A)))

 (LOCAL (DEFTHM IDEAL-CF-0-TYPE
          (IMPLIES (AND (RATIONALP A)
                        (RATIONALP B)
                        (RATIONALP C) (RATIONALP X))
                   (RATIONALP (IDEAL-CF-0 A B C X)))
          :hints
          (("Goal" :in-theory (enable IDEAL-CF-0)))))

;; Cone cofactors

 (LOCAL (DEFUND CONE-CF-0 (A B C X)
          (SQUARE (+ (* 2 (* A X)) B))))

 (LOCAL (DEFTHM CONE-CF-0-TYPE
          (IMPLIES (AND (RATIONALP A)
                        (RATIONALP B)
                        (RATIONALP C) (RATIONALP X))
                   (RATIONALP (CONE-CF-0 A B C X)))
          :hints
          (("Goal" :in-theory (enable CONE-CF-0)))))

 (LOCAL (DEFTHM CONE-CF-0-PSD
          (IMPLIES (AND (NOT (GOAL A B C X))
                        (RATIONALP A)
                        (RATIONALP B)
                        (RATIONALP C) (RATIONALP X))
                   (>= (CONE-CF-0 A B C X) 0))
          :hints
          (("Goal" :in-theory
```

```
                  (enable CONE-CF-0 PROB-0 PROB-1)))
         :rule-classes (:linear)))

;; Monoid cofactors

 (LOCAL (DEFUND MONOID-CF-0 (A B C X)
         (- 0 (- (* B B) (* 4 (* A C))))))

;; Positivstellensatz certificate

 (LOCAL (DEFUN CERT (A B C X)
         (+ (MONOID-CF-0 A B C X)
            (CONE-CF-0 A B C X)
            (* (IDEAL-CF-0 A B C X) (PROB-0 A B C X)))))

;; Contradictory results on the sign of the certificate

 (LOCAL (DEFTHMD CERT-KEY
         (IMPLIES (AND (RATIONALP A)
                       (RATIONALP B)
                       (RATIONALP C) (RATIONALP X))
                  (= (CERT A B C X) 0))
         :hints
         (("Goal" :in-theory
                  (enable SQUARE
                          CERT
                          PROB-0
                          PROB-1
                          IDEAL-CF-0 CONE-CF-0 MONOID-CF-0)))))

 (LOCAL (DEFTHM CERT-CONTRA-M-0
         (IMPLIES (AND (NOT (GOAL A B C X))
                       (RATIONALP A)
                       (RATIONALP B)
                       (RATIONALP C) (RATIONALP X))
                  (> (MONOID-CF-0 A B C X) 0))
         :hints
         (("Goal" :in-theory
                  (enable SQUARE
                          CERT
                          PROB-0
                          PROB-1
                          IDEAL-CF-0 CONE-CF-0 MONOID-CF-0)))
         :rule-classes (:linear)))

 (LOCAL (DEFTHM CERT-CONTRA-C-0
```

```
          (IMPLIES (AND (NOT (GOAL A B C X))
                        (RATIONALP A)
                        (RATIONALP B)
                        (RATIONALP C) (RATIONALP X))
                   (>= (CONE-CF-0 A B C X) 0))
          :rule-classes (:linear)))

 (LOCAL (DEFTHM CERT-CONTRA-I-0
          (IMPLIES (AND (NOT (GOAL A B C X))
                        (RATIONALP A)
                        (RATIONALP B)
                        (RATIONALP C) (RATIONALP X))
                   (= (* (IDEAL-CF-0 A B C X)
                         (PROB-0 A B C X))
                      0))
          :hints
          (("Goal" :in-theory
                   (enable SQUARE
                           CERT
                           PROB-0
                           PROB-1
                           IDEAL-CF-0 CONE-CF-0 MONOID-CF-0)))
          :rule-classes (:linear)))

 (LOCAL (DEFTHM CERT-CONTRA
          (IMPLIES (AND (NOT (GOAL A B C X))
                        (RATIONALP A)
                        (RATIONALP B)
                        (RATIONALP C) (RATIONALP X))
                   (NEQ (CERT A B C X) 0))
          :rule-classes nil))

;; Main lemma

 (LOCAL (DEFTHM MAIN
          (IMPLIES (AND (RATIONALP A)
                        (RATIONALP B)
                        (RATIONALP C) (RATIONALP X))
                   (GOAL A B C X))
          :hints
          (("Goal" :in-theory
                   (disable GOAL)
                   :use (CERT-KEY CERT-CONTRA)))
          :rule-classes nil))

;; Final theorem
```

```
(DEFTHM FINAL
  (IMPLIES (AND (RATIONALP A)
                (RATIONALP B)
                (RATIONALP C)
                (RATIONALP X) (= (+ (* A X X) (* B X) C) 0))
           (>= (- (* B B) (* 4 A C)) 0))
  :hints
  (("Goal" :in-theory
           (enable GOAL PROB-0 PROB-1) :use (MAIN)))
  :rule-classes nil))
```

3 Mathematical Background

The general setting for nonlinear real arithmetic is the theory of real closed fields (RCF). A real closed field is a field elementarily equivalent to $\mathbb{R}$ w.r.t. the language of ordered rings, i.e., the first-order language of polynomial equations and inequalities over $\mathbb{Q}[\vec{x}]$. RCF is complete, decidable and admits effective elimination of quantifiers [16, 12].

Though decidable, RCF is fundamentally infeasible. For example, Davenport-Heintz have isolated a family of n-variable RCF formulas of length $O(n)$ whose only quantifier-free equivalents must contain polynomials of degree $2^{2^{\Omega(n)}}$ and of length $2^{2^{\Omega(n)}}$ [4]. Tarski was the first to give an RCF quantifier elimination algorithm [16] but its non-elementary complexity makes it impractical for real-world use. Collins's CAD [3] achieves an asymptotic best-case of doubly-exponential complexity and is the foundation of many best performing proof procedures available in computer algebra systems and SMT solvers [12]. Nevertheless, CAD relies on complex algebro-geometric computations and to date no one has succeeded in extracting foundationally checkable proof objects from CAD.

For restricted fragments of RCF, we can do better. The purely existential fragment is known to only have singly exponential worst-case complexity [2], and convex optimization techniques can efficiently handle many specialized but practically useful classes of problems [10]. It is in this context that our work takes place: we are working only over the purely universal (dually, purely existential) fragment, and our proof construction uses convex optimization to search over a space of possible foundational proofs.

3.1 The Krivine-Stengle Positivstellensatz

The core of our proof construction relies on the Krivine-Stengle Positivstellensatz. Like its complex algebro-geometric sibling the Nullstellensatz, the Positivstellensatz guarantees the existence of algebraic proof certificates witnessing unsatisfiability. While the Nullstellensatz deals only with equations and ideals and their relationship with satisfiability over $\mathbb{C}$, the Positivstellensatz is more intricate as it must also take into account ordering relations given $\mathbb{R}$'s status as an ordered field.

Theorem 3.1 (Krivine-Stengle Positivstellensatz)**.**

$$\left(\bigwedge_i^{k_0} p_i = 0\right) \wedge \left(\bigwedge_i^{k_1} q_i \geq 0\right) \wedge \left(\bigwedge_i^{k_2} r_i \neq 0\right) \quad s.t. \quad p_i, q_i, r_i \in \mathbb{Q}[\vec{x}]$$

is unsatisfiable over $\mathbb{R}$ *iff*

$$\exists \mathrm{P} \in Ideal(p_1, \ldots, p_{k_0})$$

$$\exists \mathrm{Q} \in \mathit{Cone}(q_1, \ldots, q_{k_1})$$

$$\exists \mathrm{R} \in \mathit{Monoid}(r_1, \ldots, r_{k_2})$$

s.t.

$$\mathrm{P} + \mathrm{Q} + \mathrm{R}^2 = 0$$

where

$$\mathit{Ideal}(a_1, \ldots, a_m) = \left\{ \sum_{i=1}^{m} a_i b_i \mid b_i \in \mathbb{Q}[\vec{x}] \right\}$$

$$\mathit{Cone}(a_1, \ldots, a_m) = \left\{ r + \sum_{i=1}^{m} t_i u_i \mid r, t_i \in \sum (\mathbb{Q}[\vec{x}])^2, u_i \in \mathit{Monoid}(a_1, \ldots, a_m) \right\}$$

$$\mathit{Monoid}(a_1, \ldots, a_m) = \left\{ \prod_{i=1}^{m} (a_i)^j \mid j \in \mathbb{N} \right\}$$

$$\sum (\mathbb{Q}[\vec{x}])^2 = \left\{ \sum_{i=1}^{v} (p_i)^2 \mid p_i \in \mathbb{Q}[\vec{x}] \ \wedge \ v \in \mathbb{N} \right\}.$$

The sum $\mathrm{P} + \mathrm{Q} + \mathrm{R}^2$ is the certificate of unsatisfiability. Like we reasoned in the introduction, it is easy to see why unsatisfiability follows: appealing to the fact that ideals generalize nullity, cones generalize non-negativity, and multiplicative monoids generalize non-nullity, our constraints imply that $\mathrm{P} = 0$, $\mathrm{Q} \geq 0$ and $\mathrm{R}^2 > 0$, and thus that $\mathrm{P} + \mathrm{Q} + \mathrm{R}^2 > 0$. But by polynomial arithmetic alone $\mathrm{P} + \mathrm{Q} + \mathrm{R}^2$ reduces to 0. Thus our constraint system implies $0 > 0$ and must be unsatisfiable. The miracle of the theorem is that these certificates always exist. The next question is: how to find them?

3.2 Sums of Squares Decompositions and Semidefinite Programming

From the guise of logic, the Positivstellensatz gives us both a proof system and a completeness theorem. The original proofs establishing the Positivstellensatz, however, were non-constructive, giving no hint as to how one can effectively find the promised proofs.

A major advance occurred in 2000, with Parrilo's use of semidefinite programming (SDP) relaxations to efficiently search over convex spaces of certificate coefficients [9, 10].

From our perspective, Parillo's key theorem is the following (Theorem 5.1 of [9]):

Theorem (SDP for Positivstellensatz Search)**.** *Consider a system of polynomial equalities and inequalities. Then, the search for bounded degree Positivstellensatz refutations can be done using semidefinite programming. If the degree bound is chosen to be large enough, then the SDPs will be feasible, and the certificates obtained from its solution.*

The critical fact is that these searches are over a convex space, and thus can take advantage of efficient optimization methods. How to make the space convex? For a given certificate bound, we consider which monomials could possibly appear in the certificate, and introduce fresh variables for them. Then, the problem polynomials can be expressed as a quadratic form in the fresh variables, and a linear constraint system (*modulo* a PSD constraint on the matrix of the quadratic form) can be extracted by comparing coefficients. But optimizing linear constraints modulo a PSD matrix is a convex optimization problem: this is precisely the domain of semidefinite programming.

4 Examples, Caveats and Limitations

While we believe our present approach is promising and useful in many ways, especially for relatively small but algebraically nontrivial inequalities arising in verification practice, it is not a panacea. First, the worst-case degree bounds on certificates are in general hyper-exponential in dimension, and we experience this in practice: the more variables there are, the harder things tend to get. Second, the space of possible certificates grows rapidly as degree bounds are expanded. And third, efficient SDP solvers use numerical methods based on floating point, and it is not always easy to recover exact rational coefficients from SDP solutions. Harrison's `REAL_SOS` tactic [5] in HOL-Light addresses many of these challenges, and we refer the reader to his work for more details. Subsequent theoretical analyses have shown that some of these issues are insurmountable with the present approach [7].

Nevertheless, we are encouraged by the present state of the method. For example, the following are all problems which can be solved by Imandra, translated into ACL2, and checked successfully by ACL2 in (at most) seconds:

```
(IMPLIES (= (+ (* X X) (* Y Y) (* Z Z)) 1)
         (<= (* (+ X Y Z) (+ X Y Z)) 3))

(IMPLIES (= (+ (* W W) (* X X) (* Y Y) (* Z Z)) 1)
         (<= (* (+ W X Y Z) (+ W X Y Z)) 4))

(IMPLIES (AND (<= 0 X) (<= 0 Y) (= (* X Y) 1))
         (<= (+ X Y) (+ (* X X) (* Y Y))))

(IMPLIES (AND (>= X 1) (>= Y 1))
         (>= (* X Y) (- (+ X Y) 1)))

(IMPLIES (AND (<= 0 X) (<= 0 Y))
              (<= (* X Y (EXPT (+ X Y) 2))
                  (EXPT (+ (* X X) (* Y Y)) 2)))

(IMPLIES (AND (<= 0 A) (<= 0 B) (<= 0 C)
              (<= (* C (EXPT (+ (* 2 A) B) 3)) (* 27 X)))
          (<= (* C A A B) X))
```

There are some problems which, e.g., Harrison's `REAL_SOS` can handle, but we cannot. We are not sure why, but we conjecture this may have to do with numerical differences in the execution of the SDP solver, as SDP floating point results can be platform dependent [5]. These include:

```
(IMPLIES (AND (= (+ (* A X X X) (+ B X X) (+ C X) D) 0)
              (= (+ (* A Y Y Y) (+ B Y Y) (+ C Y) D) 0)
              (< (+ (- (* 18 A B C D) (* 4 B B D))
                    (- (* B B C C) (* 4 A C C C))
                    (- 0 (* 27 A A D D)))
                 0))
         (= X Y))
```

and

```
(IMPLIES (AND (= (- X2 U3) 0)
              (= (* (- (- X1 U1) U3) (* X2 U2)) 0)
              (= (- (* X4 X1) (* X3 U3)) 0)
              (= (- (* X4 (- U2 U1)) (* (- X3 U1) U3)) 0))
         (= (+ (- (- (* X1 X1) (* 2 X1 X3)) (* 2 X4 X2)) (* X2 X2)) 0))
```

Encouragingly, in all such failing cases, we fail even to construct a proof in Imandra, rather than finding a certificate but failing in extracting a valid ACL2 proof. In all of our current examples, if we find a certificate, we successfully construct an ACL2 version which ACL2 checks quickly.

5 Related Work

Harrison's HOL-Light `REAL_SOS` tactic [5] is the moral foundation of this work. For the case of Positivstellensatz proofs, we have in many ways simply adapted his ideas to the setting of Imandra and ACL2, including his OCaml interface to the `csdp` [1] SDP solver and techniques for rational certificate recovery. Harrison's work is based on Parillo's key insight of reducing Positivstellensatz searches to a sequence of SOS decompositions [10], which in turn builds on the Powers-Wörmann algorithm for reducing SOS decompositions to a sequence of convex SDP searches [14, 13].

6 Conclusion and Future Work

We have presented an integration of Imandra and ACL2 for constructing ACL2 proofs of nonlinear inequalities. The approach is built around the Positivstellensatz and uses convex optimization to search for foundational proofs of unsatisfiability. This work is in many ways an Imandra and ACL2 adaptation of the pioneering work of Harrison and his `REAL_SOS` tactic in HOL-Light, and further of Parrilo's work on reducing Positivstellensatz searches to semidefinite programming. We are next focusing on integrating Imandra's real algebraic counterexample search and region decomposition methods into the procedure [11, 8], and further handling problems with more general boolean structure. We also aim to develop an ACL2 client (available in, e.g., Emacs) which makes it easy to send problems to an Imandra service in the cloud and to then incorporate the delivered proofs into local developments.

References

[1] Brian Borchers (1999): *CSDP, A C library for semidefinite programming*. *Optimization Methods and Software* 11(1-4), pp. 613–623, doi:10.1080/10556789908805765. arXiv:https://doi.org/10.1080/10556789908805765.

[2] J. Canny (1993): *Improved Algorithms for Sign Determination and Existential Quantifier Elimination*. *The Computer Journal* 36(5), pp. 409–418, doi:10.1093/comjnl/36.5.409. arXiv:https://academic.oup.com/comjnl/article-pdf/36/5/409/1105564/360409.pdf.

[3] George E. Collins (1975): *Quantifier elimination for real closed fields by cylindrical algebraic decomposition*. In H. Brakhage, editor: *Automata Theory and Formal Languages*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 134–183, doi:10.1007/3-540-07407-4_17.

[4] James H. Davenport & Joos Heintz (1988): *Real quantifier elimination is doubly exponential*. *Journal of Symbolic Computation* 5(1), pp. 29–35, doi:10.1016/S0747-7171(88)80004-X. Available at `https://www.sciencedirect.com/science/article/pii/S074771718880004X`.

[5] John Harrison (2007): *Verifying Nonlinear Real Formulas Via Sums of Squares*. In Klaus Schneider & Jens Brandt, editors: *Theorem Proving in Higher Order Logics*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 102–118, doi:10.1007/978-3-540-74591-4_9.

[6] J. Krivine (1964): *Anneaux préordonnés*. *Journal d Analyse Mathématique* 12, pp. 307–326, doi:10.1007/BF02807438.

[7] David Monniaux & Pierre Corbineau (2011): *On the Generation of Positivstellensatz Witnesses in Degenerate Cases*. In Marko van Eekelen, Herman Geuvers, Julien Schmaltz & Freek Wiedijk, editors: *Interactive Theorem Proving*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 249–264, doi:10.1007/978-3-642-22863-6_19.

[8] Leonardo de Moura & Grant Olney Passmore (2013): *Computation in Real Closed Infinitesimal and Transcendental Extensions of the Rationals*. In Maria Paola Bonacina, editor: *Automated Deduction – CADE-24*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 178–192, doi:10.1007/978-3-642-38574-2_12.

[9] Pablo Parrilo (2000): *Structured Semidefinite Programs and Semialgebraic Geometry Methods in Robustness and Optimization*. *PhD thesis, Caltech*, doi:10.7907/2K6Y-CH43.

[10] Pablo A. Parrilo (2003): *Semidefinite programming relaxations for semialgebraic problems*. *Mathematical Programming* 96(2), pp. 293–320, doi:10.1007/s10107-003-0387-5.

[11] Grant Passmore, Simon Cruanes, Denis Ignatovich, Dave Aitken, Matt Bray, Elijah Kagan, Kostya Kanishev, Ewen Maclean & Nicola Mometto (2020): *The Imandra Automated Reasoning System (System Description)*. In: *Proc. 10th Int. Joint Conf. Automated Reasoning (IJCAR)*, pp. 464–471, doi:10.1007/978-3-030-51054-1_30.

[12] Grant Olney Passmore (2011): *Combined Decision Procedures for Nonlinear Arithmetics, Real and Complex*. Ph.D. thesis, University of Edinburgh.

[13] Grant Olney Passmore (2011): *Sums of Squares Methods Explained: Part I*. Technical Report, University of Cambridge, `https://www.cl.cam.ac.uk/~gp351/sos.pdf`.

[14] Victoria Powers & Thorsten Wörmann (1998): *An algorithm for sums of squares of real polynomials*. *Journal of Pure and Applied Algebra* 127(1), pp. 99–104, doi:10.1016/S0022-4049(97)83827-3. Available at `https://www.sciencedirect.com/science/article/pii/S0022404997838273`.

[15] Gilbert Stengle (1974): *A nullstellensatz and a positivstellensatz in semialgebraic geometry*. *Mathematische Annalen* 207(2), pp. 87–97, doi:10.1007/BF01362149.

[16] Alfred Tarski (1945): *A Decision Method for Elementary Algebra and Geometry*. Rand Corporation, USA.

Verification of a Rust Implementation of Knuth's Dancing Links using ACL2

David S. Hardin
Cedar Rapids, IA USA
david.s.hardin@gmail.com

"Dancing Links" connotes an optimization to a circular doubly-linked list data structure implementation which provides for fast list element removal and restoration. The Dancing Links optimization is used primarily in fast algorithms to find exact covers, and has been popularized by Knuth in Volume 4B of his seminal series *The Art of Computer Programming*. We describe an implementation of the Dancing Links optimization in the Rust programming language, as well as its formal verification using the ACL2 theorem prover. Rust has garnered significant endorsement in the past few years as a modern, memory-safe successor to C/C++ at companies such as Amazon, Google, and Microsoft, and is being integrated into both the Linux and Windows operating system kernels. Our interest in Rust stems from its potential as a hardware/software co-assurance language, with application to critical systems. We have crafted a Rust subset, inspired by Russinoff's Restricted Algorithmic C (RAC), which we have imaginatively named Restricted Algorithmic Rust, or RAR. In previous work, we described our initial implementation of a RAR toolchain, wherein we simply transpile the RAR source into RAC. By so doing, we leverage a number of existing hardware/software co-assurance tools with a minimum investment of time and effort. In this paper, we describe the RAR Rust subset, describe our improved prototype RAR toolchain, and detail the design and verification of a circular doubly-linked list data structure employing the Dancing Links optimization in RAR, with full proofs of functional correctness accomplished using the ACL2 theorem prover.

1 Introduction

The exact cover problem [17], in its simplest form, attempts to find, for an $n \times m$ matrix with binary elements, all of the subsets of the rows of the matrix such that all the column sums are exactly one. This basic notion naturally extends to matrix elements that are in some numerical range; indeed, the popular puzzle game Sudoku is an extended exact cover problem for a 9×9 matrix with element values in the range of 1 to 9, inclusive.

The exact cover problem is NP-complete, but computer scientists have devised recursive, nondeterministic backtracking algorithms to find exact covers. One such procedure is Knuth's Algorithm X, described in [17]. In this algorithm, elements of the matrix are connected via circular doubly-linked lists, and individual elements are removed, or restored, as the algorithm proceeds, undergoing backtracking, etc. As these removals and restorations out of/into the list are quite common, making these operations efficient is a laudable goal. This is where Knuth's "Dancing Links" comes in, resulting in an optimized algorithm for finding exact covers which Knuth calls DLX (Dancing Links applied to algorithm X).

2 Dancing Links

The concept behind Dancing Links is quite simple: when a given element Y of a list is removed in an exact cover algorithm, it is very likely that this same element will later be restored. Thus, rather

A. Coglio, S. Swords (Eds.): 18th ACL2 Workshop (ACL2-2023)
EPTCS 393, 2023, pp. 161–174, doi:10.4204/EPTCS.393.13

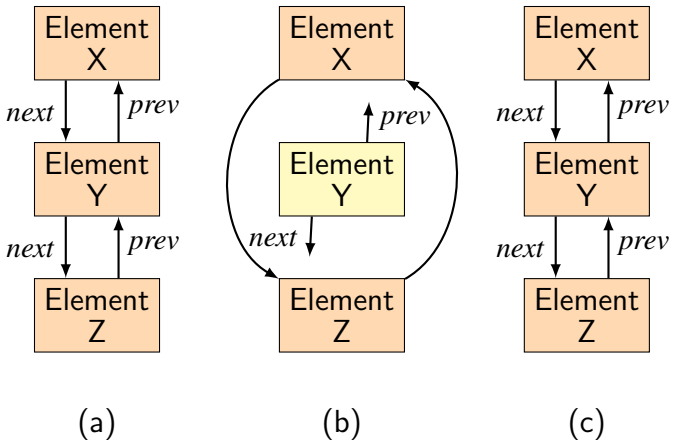


Figure 1: Dancing Links in action. (a) Portion of a circular doubly-linked list prior to a remove operation; (b) After the remove operation on element Y; (c) After the restore operation for element Y.

than "zero out" the 'previous' and 'next' links associated with element Y, as good programming hygiene would normally dictate, in Dancing Links, the programmer leaves the link values in place for the removed element. The Dancing Links `remove` operator thus deletes element Y from the list, setting the 'next' element of the preceding element X to the following element Z, and setting the 'previous' element of Z to a link to X, but not touching the 'next' and 'previous' links of the removed element Y. Later on, if Y needs to be restored, it is simply hooked back in to the list using a simple `restore` operator. In Knuth's words, if one monitors the list links as the DLX algorithm proceeds, the links appear to 'dance', hence the name. Knuth's Dancing Links functionality is summarized in Fig. 1.

3 The Rust Programming Language

The Rust programming language has garnered significant interest and use as a modern, type-safe, memory-safe, and potentially formally analyzable programming language. Google [29] and Amazon [25] are major Rust adopters, and Linus Torvalds has commented positively on the near-term ability of the Rust toolchain to be used in Linux kernel development [1]. And after spending decades dealing with a never-ending parade of security vulnerabilities due to C/C++, which continue to manifest at a high rate [24] despite their use of sophisticated C/C++ analysis tools, Microsoft announced at its BlueHat 2023 developer conference that is was beginning to rewrite core Windows libraries in Rust [6].

Our interest in Rust stems from its potential as a hardware/software co-assurance language. This interest is motivated in part by emerging application areas, such as autonomous and semi-autonomous platforms for land, sea, air, and space, that require sophisticated algorithms and data structures, are subject to stringent accreditation/certification, and encourage hardware/software co-design approaches. (For an unmanned aerial vehicle use case illustrating a formal methods-based systems engineering environment, please consult [7] [23].) In this paper, we explore the use of Rust as a High-Level Synthesis (HLS) language [26].

HLS developers specify the high-level abstract behavior of a digital system in a manner that omits hardware design details such as clocking; the HLS toolchain is then responsible for "filling in the details" to produce a Register Transfer Level (RTL) structure that can be used to realize the design in hardware.

HLS development is thus closer to software development than traditional hardware design in Hardware Description Languages (HDLs) such as Verilog or VHDL. Most incumbent HLS languages are a subset of C, e.g. Mentor Graphics' Algorithmic C [21], or Vivado HLS by Xilinx [31], although other languages have also been used, e.g. OCaml [15]. A Rust-based HLS would bring a single modern, type-safe, and memory-safe expression language for both hardware and software realizations, with very high assurance.

For formal methods researchers, Rust presents the opportunity to reason about application-level logic written in the imperative style favored by industry, but without the snarls of the unrestricted pointers of C/C++. Much progress has been made to this end in recent years, to the point that developers can verify the correctness of common algorithm and data structure code that utilizes common idioms such as records, loops, modular integers, and the like, and verified compilers can guarantee that such code is compiled correctly to binary [18]. Particular progress has been made in the area of hardware/software co-design algorithms, where array-backed data structures are common [10, 11]. (NB: This style of programming also addresses one of the shortcomings of Rust, namely its lack of support for cyclic data structures.)

As a study of the suitability of Rust as an HLS, we have crafted a Rust subset, inspired by Russinoff's Restricted Algorithmic C (RAC) [27], which we have imaginatively named Restricted Algorithmic Rust, or RAR [13]. In fact, in our first implementation of a RAR toolchain, we merely "transpile" (perform a source-to-source translation of) the RAR source into RAC. By so doing, we leverage a number of existing hardware/software co-assurance tools with a minimum investment of time and effort. By transpiling RAR to RAC, we gain access to existing HLS compilers (with the help of some simple C preprocessor directives, we are able to generate code for either the Algorithmic C or Vivado HLS toolchains). But most importantly for our research, we leverage the RAC-to-ACL2 translator that Russinoff and colleagues at Arm have successfully utilized in industrial-strength floating point hardware verification.

We have implemented several representative algorithms and data structures in RAR, including:

- a suite of array-backed algebraic data types, previously implemented in RAC (as reported in [10]);
- a significant subset of the Monocypher [30] modern cryptography suite, including XChacha20 and Poly1305 (RFC 8439) encryption/decryption, BLAKE2b hashing, and X25519 public key cryptography [12]; and
- a DFA-based JSON lexer, coupled with an LL(1) JSON parser. The JSON parser has also been implemented using Greibach Normal Form (previously implemented in RAC, as described in [14]).

The RAR examples created to date are similar to their RAC counterparts in terms of expressiveness, and we deem the RAR versions somewhat superior in terms of readability (granted, this is a very subjective evaluation).

In this paper, we will describe the development and formal verification of an array-based circular doubly-linked list (CDLL) data structure in RAR, including the Dancing Links optimization. Along the way, we will introduce the RAR subset of Rust, the RAR toolchain, the CDLL example, and detail our ACL2-based verification techniques, as well as the ACL2 books that we brought to bear on this example. It is hoped that this explication will convince the reader of the practicality of RAR as a high-assurance hardware/software co-design language, as well as the feasibility of the performing full functional correctness proofs of RAR code. We will then conclude with related and future work.

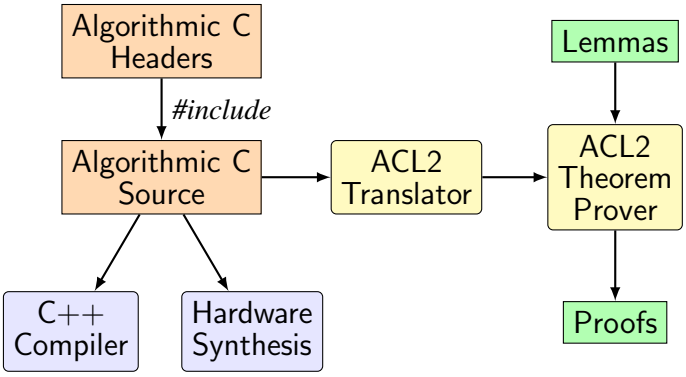


Figure 2: Restricted Algorithmic C (RAC) toolchain.

Formal Verification "Comfort Zone"	*Real-World Development*
Functional programming	Imperative programming
Total, terminating functions	Partial, potentially non-terminating functions
Non-tail-recursive functions	Loops
Okasaki-style pure functional algebraic data types	Structs, Arrays
Infinite-precision Integers, Reals	Modular Integers, IEEE 754 floating point
Linear Arithmetic	Linear and Non-linear Arithmetic
Arithmetic or Bit Vectors	Arithmetic *and* Bit Vectors

Table 1: Formal verification vs. real-world development attributes.

4 RAC: Hardware/Software Co-Assurance at Scale

In order to begin to realize hardware/software co-assurance at scale, we have conducted several experiments employing a state-of-the-art toolchain, due to Russinoff and O'Leary, and originally designed for use in floating-point hardware verification [27], to determine its suitability for the creation of safety-critical/security-critical applications in various domains. Note that this toolchain has already demonstrated the capability to scale to industrial designs in the floating-point hardware design and verification domain, as it has been used in design verifications for CPU products at both Intel and Arm.

Algorithmic C [21] is a High-Level Synthesis (HLS) language, and is supported by hardware/software co-design environments from Mentor Graphics, *e.g.*, Catapult [22]. Algorithmic C defines C++ header files that enable compilation to both hardware and software platforms, including support for the peculiar bit widths employed, for example, in floating-point hardware design.

The Russinoff-O'Leary Restricted Algorithmic C (RAC) toolchain, depicted in Fig. 2, translates a subset of Algorithmic C source to the Common Lisp subset supported by the ACL2 theorem prover, as augmented by Russinoff's Register Transfer Logic (RTL) books.

The ACL2 Translator component of Fig. 2 provides a case study in the bridging of Formal Modeling and Real-World Development concerns, as summarized in Table 1. The ACL2 translator converts imperative RAC code to functional ACL2 code. Loops are translated into tail-recursive functions, with automatic generation of measure functions to guarantee admission into the logic of ACL2 (RAC subsetting rules ensure that loop measures can be automatically determined). Structs and arrays are converted

into functional ACL2 records. The combination of modular arithmetic and bit-vector operations of typical RAC source code is faithfully translated to functions supported by Russinoff's RTL books. ACL2 is able to reason about non-linear arithmetic functions, so the usual concern about formal reasoning about non-linear arithmetic functions does not apply. Finally, the RTL books are quite capable of reasoning about a combination of arithmetic and bit-vector operations, which is a very difficult feat for most automated solvers.

Recently, we have investigated the synthesis of Field-Programmable Gate Array (FPGA) hardware directly from high-level architecture models, in collaboration with colleagues at Kansas State University. The goal of this work is to enable the generation of high-assurance hardware and/or software from high-level architectural specifications expressed in the Architecture Analysis and Design Language (AADL) [9], with proofs of correctness in ACL2.

5 Rust and RAR

The Rust Programming Language [16] is a modern, high-level programming language designed to combine the code generation efficiency of C/C++ with drastically improved type safety and memory management features. A distinguishing feature of Rust is that a non-scalar object may only have one owner. For example, one cannot assign a reference to an object in a local variable, and then pass that reference to a function. This restriction is similar to those imposed on ACL2 single-threaded objects (stobjs) [4], with the additional complexities of enforcing such "single-owner" restrictions in the context of a general-purpose, imperative programming language. The Rust runtime performs array bounds checking, as well as arithmetic overflow checking (the latter can be disabled by a build environment setting).

In most other ways, Rust is a fairly conventional modern programming language, with interfaces (called traits), lambdas (termed closures), and pattern matching, as well as a macro capability. Also in keeping with other modern programming language ecosystems, Rust features a language-specific build and package management sytem, named `cargo`.

5.1 Restricted Algorithmic Rust

As we wish to utilize the RAC toolchain as a backend in our initial work, Restricted Algorithmic Rust is semantically equivalent to RAC. Thus, we adopt the same semantic restrictions as described in Russinoff's book. Additionally, in order to enable translation to RAC, as well as to ease the transition from C/C++, RAR supports a commonly used macro that provides a C-like *for* loop in Rust. Note that, despite the restrictions, RAR code is proper Rust; it compiles to binary using the standard Rust compiler.

RAR is transpiled to RAC via a source-to-source translator, as depicted in Fig. 3. Our transpiler is based on the `plex` parser and lexer generator [28] source code. We thus call our transpiler *Plexi*, a nickname given to a famous (and now highly sought-after) line of Marshall guitar amplifiers of the mid-1960s. Plexi performs lexical and syntactic transformations that convert RAR code to RAC code. Recent improvements in the `plexi` tool include better handling of array declarations, as well as providing support for Rust `const` declarations.

The generated RAC code can then be compiled using a C/C++ compiler, fed to an HLS-based FPGA compiler, as well as translated to ACL2 via the RAC ACL2 translator, as illustrated in Fig. 3.

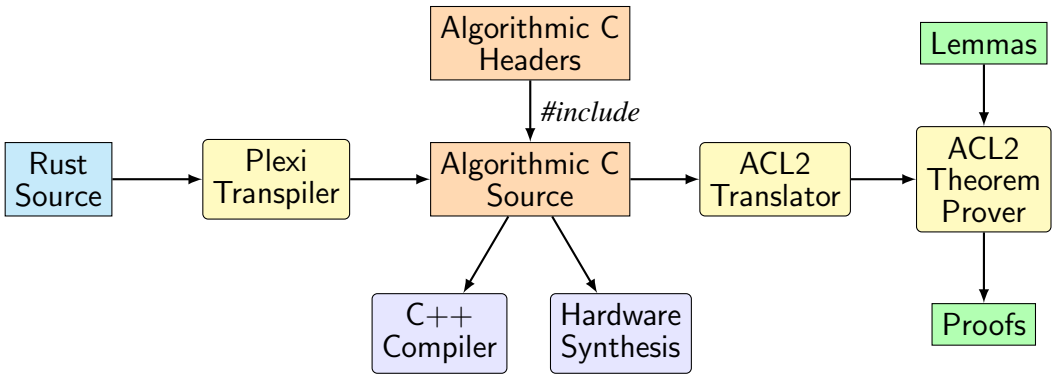


Figure 3: Restricted Algorithmic Rust (RAR) toolchain.

6 Dancing Links in Rust

In this section, we describe an array-based circular doubly-linked list (CDLL) employing Knuth's "Dancing Links" optimization, realized using our RAR Rust subset. The CDLL data structure implementation constitutes over 700 lines of Rust code, which becomes 890 lines of code when translated to ACL2.

6.1 Definitions

First, we present the basic RAR declaration for the CDLL.

```
const CDLL_MAX_NODE1: usize = 8191;
const CDLL_MAX_NODE: usize = CDLL_MAX_NODE1 - 1;

#[derive(Copy, Clone)]
struct CDLLNode {
  alloc: u2,
  val: i64,
  prev: usize,
  next: usize,
}

#[derive(Copy, Clone)]
struct CDLL {
  nodeHd: usize,
  nodeCount: usize,
  nodeArr: [CDLLNode; CDLL_MAX_NODE1],
}
```

Rust data structure declarations are similar to those in C, but struct elements are declared by specifying the element name, followed by the `:` separator, then the element type. Also note that Rust pragmas may be given using the `derive` attribute. In the declaration above, the array `nodeArr` holds the list element nodes. Each element has `next` and `prev` indices. Note that indices in Rust are normally declared to be of the `usize` type. Note also that by using array indices instead of references, we get around

```
fn CDLL_remove(n: usize, mut CDObj: CDLL) -> CDLL {
  if (n > CDLL_MAX_NODE) {
    return CDObj;
  } else {
    if (n == CDObj.nodeHd) {  // Can't remove head
      return CDObj;
    } else {
      if (CDObj.nodeCount < 3) {  // Need three elements for remove to work
        return CDObj;
      } else {
        let nextNode: usize = CDObj.nodeArr[n].next;
        let prevNode: usize = CDObj.nodeArr[n].prev;

        CDObj.nodeArr[prevNode].next = nextNode;
        CDObj.nodeArr[nextNode].prev = prevNode;

        CDObj.nodeCount = CDObj.nodeCount - 1;

        return CDObj;
      }
    }
  }
}
```

Figure 4: `cdll_remove()` function in RAR.

Rust ownership model issues with circular data structures. The `alloc` field of the `CDLLNode` structure is declared to be a two bit unsigned field, but its only allowed values are two non-zero values: 2 (not currently allocated), and 3 (allocated). The reason for this has to do with the details of ACL2 untyped record reasoning, which will be discussed in Section 6.2.

The Dancing Links operators `cdll_remove` and `cdll_restore` are presented in Figures 4 and 5, respectively. Rust functions begin with the `fn` keyword, followed by the function name, a parenthesized list of parameters, the `->` (returns) symbol, the return type name, followed by the function body (delimited by a curly brace pair). A function parameter list element consists of the parameter name, the `:` symbol, then the parameter type. Additional parameter modifiers, for example `mut`, may be present to indicate that the parameter is changed in the function body. Within the function body, the syntax is similar to other C-like languages, but local variable declarations begin with `let`, and use the variable name, `:`, variable type declaration syntax. A local variable declaration may also require the `mut` modifier if that local variable is updated after its initialization.

6.2 Translation to ACL2

We use `Plexi` to transpile the RAR source to RAC (not shown), then use the RAC translator to convert the resulting RAC source to ACL2. The translation of `cdll_restore()` appears in Fig. 6.

```
fn CDLL_restore(n: usize, mut CDObj: CDLL) -> CDLL {
  if (n > CDLL_MAX_NODE) {
    return CDObj;
  } else {
    if (n == CDObj.nodeHd) {  // Can't restore head
      return CDObj;
    } else {
      if ((CDObj.nodeCount < 2) ||        // Need two elements for restore to work
          (CDObj.nodeCount == CDLL_MAX_NODE1))  {  // Can't restore to a full list
        return CDObj;
      } else {

        let prevNode: usize = CDObj.nodeArr[n].prev;
        let nextNode: usize = CDObj.nodeArr[n].next;

        CDObj.nodeArr[prevNode].next = n;
        CDObj.nodeArr[nextNode].prev = n;

        CDObj.nodeCount = CDObj.nodeCount + 1;

        return CDObj;
      }
    }
  }
}
```

Figure 5: `cdll_restore()` function in RAR.

```
(DEFUND CDLL_RESTORE (N CDOBJ)
   (IF1 (LOG> N (CDLL_MAX_NODE))
        CDOBJ
        (IF1 (LOG= N (AG 'NODEHD CDOBJ))
             CDOBJ
             (IF1 (LOGIOR1 (LOG< (AG 'NODECOUNT CDOBJ) 2)
                           (LOG= (AG 'NODECOUNT CDOBJ)
                                 (CDLL_MAX_NODE1)))
                  CDOBJ
                  (LET* ((PREVNODE (AG 'PREV (AG N (AG 'NODEARR CDOBJ))))
                         (NEXTNODE (AG 'NEXT (AG N (AG 'NODEARR CDOBJ))))
                         (CDOBJ (AS 'NODEARR
                                    (AS PREVNODE
                                        (AS 'NEXT
                                            N (AG PREVNODE (AG 'NODEARR CDOBJ)))
                                        (AG 'NODEARR CDOBJ))
                                    CDOBJ))
                         (CDOBJ (AS 'NODEARR
                                    (AS NEXTNODE
                                        (AS 'PREV
                                            N (AG NEXTNODE (AG 'NODEARR CDOBJ)))
                                        (AG 'NODEARR CDOBJ))
                                    CDOBJ)))
                        (AS 'NODECOUNT
                            (+ (AG 'NODECOUNT CDOBJ) 1)
                            CDOBJ))))))
```

Figure 6: `cdll_restore()` function translated to ACL2 using the RAC tools.

The first thing to note about Fig. 6 is that, even though we are two translation steps away from the original RAR source, the translated function is nonetheless quite readable, which is a rare thing for machine-generated code. Another notable observation is that struct and array 'get' and 'set' operations become untyped record operators, `AG` and `AS`, respectively — these are slight RAC-specific customizations of the usual ACL2 untyped record operators. Further, `IF1` is a RAC-specific macro, and `LOG>`, `LOG=`, `LOG<`, and `LOGIOR1` are all RTL functions. Thus, much of the proof effort involved with RAR code is reasoning about untyped records and RTL — although not a lot of RTL-specific knowledge is needed, at least in our experience.

One aspect of untyped records that can be tricky is that record elements that take on the default value are not explicitly stored in the association list for the record. For RAC untyped records, that default value is zero. Thus, it is easy for a given record to attain a `nil` value. When reasoning about arrays of such records, it is often desirable to be able to state that the array size remains constant. Thus, for example, for the `CDLL` array `nodeArr` of Section 6.1, we ensure that all `CDLLNode` elements of that array are non-nil by making sure that the `alloc` fields of the `CDLLNode` elements are always non-zero (2 or 3).

6.3 Dancing Links Theorems

Once we have translated the circular doubly-linked list functions into ACL2, we can begin to prove theorems about the data structure implementation. We begin by defining a "well-formedness" predicate for CDLLs.

```
(defun cdllnodeArrp-helper (arr j)
  (cond ((not (true-listp arr)) nil)
        ((null arr) t)
        ((not (and (integerp j) (<= 0 j))) nil)
        ((not (consp (car arr))) nil)
        ((not (= (car (car arr)) j)) nil)
        ((not (cdllnodep (cdr (car arr)))) nil)
        (t (cdllnodeArrp-helper (cdr arr) (1+ j)))))

(defun cdllnodeArrp (arr)
  (cdllnodeArrp-helper arr 0))

(defun cdllp (Obj)
  (and (integerp (ag 'nodeHd Obj))
       (<= 0 (ag 'nodeHd Obj))
       (<= (ag 'nodeHd Obj) (CDLL_MAX_NODE))
       (integerp (ag 'nodeCount Obj))
       (<= 0 (ag 'nodeCount Obj))
       (<= (ag 'nodeCount Obj) (CDLL_MAX_NODE1))
       (cdllnodeArrp (ag 'nodeArr Obj))
       (= (len (ag 'nodeArr Obj)) (CDLL_MAX_NODE1))))
```

Given this definition of a good CDLL state, we can prove functional correctness theorems for Dancing Links operations, of the sort stated below. Note that this proof requires some detailed well-formedness hypotheses related to the `prev` and `next` indices for the nth element:

```
(defthm restore-of-remove--thm
 (implies
  (and (cdllp Obj)
       (good-nodep n Obj)
       (not (= n (ag 'nodeHd Obj)))
       (>= (ag 'nodeCount Obj) 3))
  (= (CDLL_restore n (CDLL_remove n Obj))
     Obj)))
```

ACL2 performs the correctness proof for this `cdll_restore` of `cdll_remove` theorem automatically. In addition to the Dancing Links operator proofs, we have proved approximately 160 theorems related to the CDLL data structure, including theorems about `cdll_cns()` (`cons` equivalent), `cdll_rst()` (`cdr` equivalent), `cdll_snc()` (add to end of data structure), `cdll_tsr()` (delete from end of data structure), `cdll_nth()`, etc. All of these proofs will be made publicly available in the ACL2 workshop books repository.

7 Related Work

A number of domain-specific languages targeting both hardware and software realization, and providing support for formal verification, have been created. Cryptol [5], for example, has been employed as a "golden spec" for the evaluation of cryptographic implementations, in which automated tools perform equivalence checking between the Cryptol spec for a given algorithm, and the VHDL implementation.

Formal verification systems for Rust include Creusot [8], based on WhyML; Prusti [3], based on the Viper verification toolchain; and RustHorn [20], based on constrained Horn clauses. AWS is developing a model-checker for Rust, Kani [2]. Additionally, Carnegie-Mellon University is developing Verus, an SMT-based tool for formally verifying Rust programs [19]. With Verus, programmers express proofs and specifications using Rust syntax, allowing proofs to take advantage of Rust's linear types and borrow checking. It will be interesting to attempt the sorts of correctness proofs achievable on our system using these verification tools.

8 Conclusion

We have developed a prototype toolchain to allow the Rust programming language to be used as a hardware/software co-design and co-assurance language for critical systems, standing on the shoulders of Russinoff's team at Arm, and all the great work they have done on Restricted Algorithmic C. We have demonstrated the ability to establish the correctness of several practical data structures commonly found in high-assurance systems (*e.g.*, array-backed singly-linked lists, doubly-linked lists, stacks, and dequeues) through automated formal verification, enabled by automated source-to-source translation from Rust to RAC to ACL2, and have detailed the specification and verification of one such data structure, a circular doubly-linked list employing Knuth's "Dancing Links" optimization. We have also successfully applied our toolchain to cryptography and data format filtering examples typical of the sorts of algorithms that one encounters in critical systems development.

In future work, we will continue to develop our toolchain, increasing the number of Rust features that we can support in the RAR subset, as well as continuing to improve the ACL2 verification libraries in order to increase the ability to discharge RAR correctness proofs automatically. We will also continue to

work with our colleagues at Kansas State University on the direct synthesis and verification of RAR code from architectural models, as well as working with colleagues at the University of Kansas on verified synthesis of Rust code from high-level attestation protocol specifications written using the Coq theorem prover.

9 Acknowledgments

Many thanks to Donald Knuth for his detailed study of exact cover problems in general, and the "Dancing Links" optimization in particular, that can now be found in Volume 4B of his seminal series, *The Art of Computer Programming*. It was a pleasure discovering this particular corner of Computer Science, beginning when the author accidentally stumbled upon a previously recorded Knuth "Christmas Lecture" on the subject in late 2022.

Previous foundational work on hardware/software co-assurance in Rust was funded by DARPA contract HR00111890001. The views, opinions and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

Many thanks to David Russinoff of Arm for developing and improving the RAC toolchain, without which most of the current work would not be possible. Thanks also go to the anonymous reviewers for their insightful comments.

References

[1] Ron Amadeo (2021): *Google is now writing low-level Android code in Rust*. Available at `https://arstechnica.com/gadgets/2021/04/google-is-now-writing-low-level-android-code-in-rust/`.

[2] Amazon Web Services (2022): *Announcing the Kani Rust Verifier Project*. Available at `https://model-checking.github.io/kani-verifier-blog/2022/05/04/announcing-the-kani-rust-verifier-project.html?fbclid=IwAR2M_B1IEBfkVhIXSuuAxt3McC_QpUnTuzDq9jG40HOaJzxw8z1Nw9XU_i4`.

[3] V. Astrauskas, A. Bílý, J. Fiala, Z. Grannan, C. Matheja, P. Müller, F. Poli & A. J. Summers (2022): *The Prusti Project: Formal Verification for Rust (invited)*. In: *NASA Formal Methods (14th International Symposium)*, Springer, pp. 88–108, doi:10.1007/978-3-031-06773-0_5. Available at `https://link.springer.com/chapter/10.1007/978-3-031-06773-0_5`.

[4] Robert S. Boyer & J Strother Moore (2002): *Single-Threaded Objects in ACL2*. In: *Practical Aspects of Declarative Languages, 4th International Symposium, PADL 2002, Portland, OR, USA, January 19-20, 2002, Proceedings*, *LNCS* 2257, Springer, pp. 9–27, doi:10.1007/3-540-45587-6_3.

[5] Sally Browning & Philip Weaver (2010): *Designing Tunable, Verifiable Cryptographic Hardware Using Cryptol*. In David S. Hardin, editor: *Design and Verification of Microprocessor Systems for High-Assurance Applications*, Springer, pp. 89–143, doi:10.1007/978-1-4419-1539-9_4.

[6] Thomas Claburn (2023): *Microsoft is busy rewriting core Windows code in memory-safe Rust*. Available at `https://www.theregister.com/2023/04/27/microsoft_windows_rust/`.

[7] Darren Cofer, Isaac Amundson, Junaid Babar, David Hardin, Konrad Slind, Perry Alexander, John Hatcliff, Robby, Gerwin Klein, Corey Lewis, Eric Mercer & John Shackleton (2022): *Cyber Assured Systems Engineering at Scale*. In: *IEEE Security & Privacy*, pp. 52–64, doi:10.1109/MSEC.2022.3151733.

[8] Xavier Denis (2022): *Creusot*. Available at `https://github.com/xldenis/creusot`.

[9] Peter H. Feiler & David P. Gluch (2012): *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*, 1st edition. Addison-Wesley Professional.

[10] David S. Hardin (2020): *Put Me on the RAC*. In: *Proceedings of the Sixteenth International Workshop on the ACL2 Theorem Prover and its Applications (ACL2-20)*, pp. 142–145, doi:10.4204/eptcs.327.13.

[11] David S. Hardin (2020): *Verified Hardware/Software Co-Assurance: Enhancing Safety and Security for Critical Systems*. In: *Proceedings of the 2020 IEEE Systems Conference*, doi:10.1109/SysCon47679.2020.9381831.

[12] David S. Hardin (2022): *Hardware/Software Co-Assurance for the Rust Programming Language Applied to Zero-Trust Architecture Development*. *ACM SIGAda Ada Letters* 42(2), pp. 55–61, doi:10.1145/3591335.3591340.

[13] David S. Hardin (2022): *Hardware/Software Co-Assurance using the Rust Programming Language and ACL2*. In: *Proceedings of the Seventeenth International Workshop on the ACL2 Theorem Prover and its Applications (ACL2-22)*, pp. 202–216, doi:10.4204/EPTCS.359.16.

[14] David S. Hardin & Konrad L. Slind (2021): *Formal Synthesis of Filter Components for Use in Security-Enhancing Architectural Transformations*. In: *Proceedings of the Seventh Workshop on Language-Theoretic Security, 42nd IEEE Symposium and Workshops on Security and Privacy (LangSec 2021)*, doi:10.1109/SPW53761.2021.00024.

[15] Jane Street Group, LLC (2023): *Hardcaml: An OCaml library for designing and testing hardware designs*. Available at https://github.com/janestreet/hardcaml.

[16] Steve Klabnik & Carol Nichols (2018): *The Rust Programming Language*. No Starch Press.

[17] Donald E. Knuth (2022): *The Art of Computer Programming*. 4B: Combinatorial Algorithms, Part 2, Addison-Wesley.

[18] Ramana Kumar, Magnus O. Myreen, Michael Norrish & Scott Owens (2014): *CakeML: a verified implementation of ML*. In Suresh Jagannathan & Peter Sewell, editors: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, ACM, pp. 179–192, doi:10.1145/2535838.2535841.

[19] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno & Chris Hawblitzel (2023): *Verus: Verifying Rust Programs Using Linear Ghost Types*. *Proc. ACM Program. Lang.* 7(OOPSLA1), doi:10.1145/3586037.

[20] Yusuke Matsushita, Takeshi Tsukada & Naoki Kobayashi (2021): *RustHorn: CHC-Based Verification for Rust Programs*. *ACM Trans. Program. Lang. Syst.* 43(4), doi:10.1145/3462205.

[21] Mentor Graphics Corporation (2016): *Algorithmic C (AC) Datatypes*. Available at https://www.mentor.com/hls-lp/downloads/ac-datatypes.

[22] Mentor Graphics Corporation (2020): *Catapult High-Level Synthesis*. Available at https://www.mentor.com/hls-lp/catapult-high-level-synthesis/.

[23] Eric Mercer, Konrad Slind, Isaac Amundson, Darren Cofer, Junaid Babar & David Hardin (2023): *Synthesizing Verified Components for Cyber Assured Systems Engineering*. In: *Software and Systems Modeling*, 22, pp. 1451–1471, doi:10.1007/s10270-023-01096-3.

[24] Matt Miller (2019): *A proactive approach to more secure code*. Available at https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/.

[25] Shane Miller & Carl Lerche (2022): *Sustainability with Rust*. Available at https://aws.amazon.com/blogs/opensource/sustainability-with-rust/.

[26] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, Jason Anderson & Koen Bertels (2016): *A Survey and Evaluation of FPGA High-Level Synthesis Tools*. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35(10), pp. 1591–1604, doi:10.1109/TCAD.2015.2513673.

[27] David M. Russinoff (2022): *Formal Verification of Floating-Point Hardware Design: A Mathematical Approach*, second edition. Springer, doi:10.1007/978-3-030-87181-9.

[28] Geoffry Song (2020): *plex: a parser and lexer generator as a Rust procedural macro*. Available at https://github.com/goffrie/plex.

[29] Jeff Vander Stoep & Stephen Hines (2021): *Rust in the Android platform*. Available at https://security.googleblog.com/2021/04/rust-in-android-platform.html.

[30] Loup Vaillant (2022): *Monocypher: Boring Crypto that Simply Works*. Available at https://monocypher.org.

[31] Xilinx, Inc. (2018): *Vivado Design Suite User Guide: High-Level Synthesis*. Available at https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug902-vivado-high-level-synthesis.pdf.