EPTCS 423

Proceedings of the

19th International Workshop on the ACL2 Theorem Prover and Its Applications

Austin, TX, 12-13 May, 2025

Edited by: Ruben Gamboa and Panagiotis Manolios

Published: 25th July 2025
DOI: 10.4204/EPTCS.423
ISSN: 2075-2180
Open Publishing Association

Table of Contents

Preface

Ruben Gamboa
University of Wyoming
ruben@uwyo.edu

Panagiotis Manolios
Northeastern University
pete@ccs.neu.edu

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.

This volume contains the proceedings of the 19th International Workshop on the ACL2 Theorem Prover and its Applications (ACL2 2025), which was held in Austin, Texas, on May 12-13, 2025. ACL2 Workshops have been held regularly since 1999, typically in 18-month intervals.

These proceedings include nine long papers and two extended abstracts, all of which were reviewed by at least three members of the program committee. The workshop also included several "rump session" talks—short unpublished presentations that discussed ongoing research—as well as two invited talks, the first from Swarat Chaudhuri of the University of Texas, and the second from Warren Hunt of the University of Texas and Anna Slobodova of Arm.

At the workshop, awards were given for best student paper. This award resulted in a tie between two papers:

- Ankit Kumar, "A Formalization of the Correctness of the Floodsub Protocol"
- Andrew Walter, "An ACL2s Interface to Z3"

As program chairs we are grateful for the other leaders of this workshop: the organizing chairs, Matt Kaufmann and Eric Smith; the arrangements chairs, Carl Kwan and Maxine Xin; and the registration chair, David Rager. We also wish to thank the program committee for all of their hard work:

- Harsh Raju Chamarthi, Rivos Inc.
- Alessandro Coglio, Kestrel Institute and Provable Inc.
- Jared Davis, Amazon.com Services LLC
- Ruben Gamboa, University of Wyoming & Kestrel Institute
- Shilpi Goel, Amazon Web Services
- David Greve
- Mark Greenstreet, The University of British Columbia
- David Hardin, Collins Aerospace
- Warren Hunt, The University of Texas at Austin
- Mitesh Jain, Rivos Inc.
- Matt Kaufmann, The University of Texas at Austin (retired)
- Mayank Manjrekar, ARM Ltd.
- Panagiotis Manolios, Northeastern University
- Eric McCarthy, Kestrel Institute and Provable Inc.

Gamboa and Manolios (Eds):
ACL2 Workshop 2025
EPTCS 423, 2025, pp. ii–iii, doi:10.4204/EPTCS.423.0

- Sandip Ray, University of Florida
- Jose-Luis Ruiz-Reina, University of Seville
- David Russinoff, ARM Ltd.
- Anna Slobodova, ARM Ltd.
- Eric Smith, Kestrel Institute
- Sudarshan Srinivasan, North Dakota State University
- Rob Sumners, Advanced Micro Devices
- Sol Swords
- Freek Verbeek, Open University of The Netherlands
- Max von Hippel, Benchify
- Bill Young, The University of Texas at Austin

Ruben Gamboa and Panagiotis Manolios
Program chairs
May 2025

A Formalization of Elementary Linear Algebra: Part I

David M. Russinoff
david@russinoff.com

This is the first installment of an exposition of an ACL2 formalization of elementary linear algebra, focusing on aspects of the subject that apply to matrices over an arbitrary commutative ring with identity, in anticipation of a future treatment of the characteristic polynomial of a matrix, which has entries in a polyniomial ring. The main contribution of this paper is a formal theory of the determinant, including its characterization as the unique alternating n-linear function of the rows of an $n \times n$ matrix, multiplicativity of the determinant, and the correctness of cofactor expansion.

1 Introduction

This is the first installment of an exposition of an ACL2 formalization of elementary linear algebra, covering the basic algebra of matrices and the theory of determinants. Part II [14], also included in this workshop, addresses row reduction and its application to matrix invertibility and simultaneous systems of linear equations. Additional topics to be covered in future installments include vector spaces, linear transformations, polynomials, eigenvectors. and diagonalization.

This ordering of topics departs from the typical syllabus of an introductory course in the subject. Most elementary linear algebra textbooks treat the solution of simultaneous linear equations in the first chapter, perhaps to reassure the student of the practical utility of the theory. Consequently, (since this process depends on the existence of a multiplicative inverse) the entries of a matrix are assumed at the outset to range over a field (often the real numbers) rather than a more general commutative ring. This assumption, however, is not required for the main results of matrix algebra or the properties of the determinant; in fact, there are numerous applications for which it does not hold [1]. Indeed, several chapters later, one finds that the theory of eigenvalues is based on the fundamental notion of the characteristic polynomial of a matrix over a field F, which is properly defined as the determinant of a matrix with entries in the polynomial ring $F[t]$. In most cases, this problem is simply ignored [5, 6, 10]. A rare exception is a comparatively rigorous treatment by Hoffman and Kunze [4], on which our formalization is partly based (and from which this author learned the subject as a college sophomore). In Chapter 5, (anticipating the introduction of the characteristic polynomial) they define the determinant of a matrix over an arbitrary commutative ring with unity and ask the reader to determine for himself which of the results of the preceding chapters, though stated and proved for matrices over a field, apply more generally to commutative rings.

Neither of these strategies will serve our purpose. Unconstrained by pedagogical considerations, we pursue a more principled development, separating those aspects of the theory that are valid for general commutative rings from those that depend on the existence of a multiplicative inverse. The former topics are treated in this paper; the latter in Part II. All supporting proof scripts reside in the shared ACL2 directory `books/projects/linear/`.

In Section 2, we introduce the notion of an abstract commutative ring with unity by means of an encapsulated set of constrained functions and associated theorems corresponding to the standard ring axioms. Section 3 covers the algebra of matrices and the transpose operator. The main contribution of this paper is a formal theory of determinants, based on the classical definition, which appeals to the properties

Gamboa and Manolios (Eds):
ACL2 Workshop 2025
EPTCS 423, 2025, pp. 1–18, doi:10.4204/EPTCS.423.1

of the symmetric group. This consideration was a factor in our broader plan of a formalization of algebra beginning with the theory of finite groups [11, 12, 13], on which the present work critically depends. In Section 4, we define the determinant and derive its main properties, including its uniqueness as an alternating n-linear function of the rows of an $n \times n$ matrix. Multiplicativity is derived as a consequence of this result, which is further exploited in Section 5 to establish the correctness of cofactor expansion and the properties of the classical adjoint. The proofs of uniqueness and its consequences illustrate the use of encapsulation and functional instantiation as a substitute for higher order logical reasoning in the first order logic of ACL2.

Previous work on matrix algebra within the ACL2 community includes a formalization by Gamboa et al. based on ACL2 arrays [2], another by Hendrix with matrices defined as simple list structures [3], and Kwan's proofs of correctness of several numerical algorithms [7, 8]. Our matrix representation scheme is essentially that of Hendrix (which was also adopted by Kwan), but since we require entries ranging over an abstract ring rather than the `acl2-number` type, ours is constructed independently. Only the first of these cited references provides a general definition of the determinant, with no proofs of its properties. A variety of linear algebra formalizations have been based on other theorem provers [9, 15, 16, 17, 18], but we are not aware of any that has produced the full list of results reported above.

2 Commutative Rings

In `ring.lisp`, the axioms of a commutative ring with unity are formalized by an encapsulation, partially displayed below:

```
(encapsulate (((rp *) => *)                    ;ring element recognizer
              ((r+ * *) => *) ((r* * *) => *) ;addition and multiplication
              ((r0) => *) ((r1) => *)          ;identities
              ((r- *) => *))                   ;additive inverse
  (local (defun rp (x) (rationalp x)))
  (local (defun r+ (x y) (+ x y)))
  (local (defun r* (x y) (* x y)))
  (local (defun r0 () 0))
  (local (defun r1 () 1))
  (local (defun r- (x) (- x)))
  ;; Closure:
  (defthm r+closed (implies (and (rp x) (rp y)) (rp (r+ x y))))
  (defthm r*closed (implies (and (rp x) (rp y)) (rp (r* x y))))
  ;; Commutativity:
  ...
}
```

This introduces six constrained functions: `rp` is a predicate that recognizes an element of the ring; `r+` and `r*` are the binary addition and multiplication operations; the constants `(r0)` and `(r1)` are the identity elements of these operations, respectively; and `r-` is the unary addition inverse. Note that these functions are locally defined to be the corresponding functions pertaining to the rational numbers (an arbitrary choice—the recognizer `(integerp x)` would have worked just as well as `(rationalp x)`). The exported theorems (mostly omitted above) are the usual ring axioms: closure, commutativity, and associativity of both operations; properties of the identities and the additive inverse; and the distributive law. Informally, we shall refer to the ring `R` that is characterized by these axioms, and elements of `R` are sometimes called *scalars*. When our intention is clear, we may abbreviate `(r0)` and `(r1)` as 0 and 1, respectively.

The file also contains some trivially derived variants of the axioms, along with definitions of several functions pertaining to lists of ring elements and proofs of their basic properties:

- `rlistp` is a predicate that recognizes a vector, i.e., a proper list of scalars, which we call an *rlist*;
- `rlistnp` recognizes an rlist of a specified length;
- `rlist0p` recognizes an rlist of which every member is `(r0)`;
- `rlistn0` returns an rlist of a specified length of which every member is `(r0)`;
- `rlist-sum` and `rlist-prod` compute the sum and product, respectively, of the members of an rlist;
- `rlist-scalar-mul` multiplies each member of an rlist by a given scalar and returns a list of the products;
- `rdot` computes the *dot product* of two rlists of the same length, i.e., the sum of the products of corresponding members;
- `rdot-list` returns the list of dot products of an rlist with the members of a list of rlists.

The reader may anticipate that a function name containing the character `r`, suggesting *ring*, is likely to have an analog in Part II with `r` replaced by `f`, suggesting *field*.

3 Matrices

The ACL2 events reported in this section are taken from the file `rmat.lisp`, which begins with the definition of an `m`×`n` *matrix* `a` over the ring `R` as a proper list of `m` rlists, each of length `n`:

```
(defun rmatp (a m n)
  (if (zp m)
      (null a)
    (and (consp a)
         (rlistnp (car a) n)
         (rmatp (cdr a) (1- m) n))))
```

Each member of `a` is a *row*; a *column* is constructed by extracting an entry from each row:

```
(defun row (i a) (nth i a))
(defun col (j a)
  (if (consp a)
      (cons (nth j (car a)) (col j (cdr a)))
    ()))
```

The entry of `a` in row `i` and column `j`:

```
(defun entry (i j a) (nth j (nth i a)))
```

The basic operation of replacing row `k` of `a` with an rlist `r`:

```
(defun replace-row (a k r)
  (if (zp k)
      (cons r (cdr a))
    (cons (car a) (replace-row (cdr a) (1- k) r))))
```

If two m×n matrices are not equal, then some pair of corresponding entries are different. The function entry-diff conducts a search and returns the row and column in which this occurs:

```
(defthmd rmat-entry-diff-lemma
  (implies (and (posp m) (posp n) (rmatp a m n) (rmatp b m n) (not (equal a b)))
           (let* ((pair (entry-diff a b)) (i (car pair)) (j (cdr pair)))
             (and (natp i) (< i m) (natp j) (< j n)
                  (not (equal (entry i j a) (entry i j b)))))))
```

If we can prove that corresponding entries of a and b are equal, then we may invoke this result to conclude that a = b.

The recursive definitions of the sum of two matrices, (rmat-add a b), and the product of a scalar and a matrix, (rmat-scalar-mul c a), are trival. We shall also find it convenient to define the sum of the entries of a matrix in row-major order:

```
(defun rmat-sum (a)
  (if (consp a)
      (r+ (rlist-sum (car a)) (rmat-sum (cdr a)))
    (r0)))
```

Matrix multiplication is a more complicated operation, deferred to Subsection 3.2.

3.1 Transpose

The *transpose* of a matrix is the list of its columns:

```
(defun transpose-mat-aux (a j n)
  (if (and (natp j) (natp n) (< j n))
      (cons (col j a) (transpose-mat-aux a (1+ j) n))
    ()))
(defund transpose-mat (a) (transpose-mat-aux a 0 (len (car a))))
```

We list some simple consequences of the definition:

```
(defthm transpose-rmat-entry
  (implies (and (posp m) (posp n) (rmatp a m n) (natp j) (< j n) (natp i) (< i m))
           (equal (entry j i (transpose-mat a))
                  (entry i j a))))
(defthm transpose-rmat-2
  (implies (and (posp m) (posp n) (rmatp a m n))
           (equal (transpose-mat (transpose-mat a))
                  a)))
(defthmd col-transpose-rmat
  (implies (and (posp m) (posp n) (rmatp a m n) (natp j) (< j m))
           (equal (col j (transpose-mat a))
                  (row j a))))
```

The replacement of a column is now readily defined using the transpose:

```
(defund replace-col (a k r) (transpose-mat (replace-row (transpose-mat a) k r)))
```

Our proof of associativity of matrix multiplication uses the observation that the entries of an m×n matrix a have the same sum, as computed by rmat-sum, as those of its transpose. This is trivially true if either m or n is 0. Otherwise, we derive the (m-1)x(n-1) matrix (strip-mat a) by deleting the first row and the first column of a, and prove the following:

```
(defthmd sum-rmat-strip-mat
  (implies (and (posp m) (posp n) (rmatp a m n))
           (equal (rmat-sum a)
                  (r+ (entry 0 0 a)
                      (r+ (r+ (rlist-sum (cdr (row 0 a)))
                              (rlist-sum (cdr (col 0 a))))
                          (rmat-sum (strip-mat a)))))))
```

The desired lemma follows by induction, using `sum-rmat-strip-mat` to rewrite both sides of the equation and `col-transpose-rmat` to complete the proof:

```
(defthmd sum-rmat-transpose
  (implies (and (natp m) (natp n) (rmatp a m n))
           (equal (rmat-sum (transpose-mat a))
                  (rmat-sum a))))
```

3.2 Multiplication

The product of matrices `a` and `b` is defined when the number of columns of `a` is the number of rows of `b`. The product has the same number of rows as `a` and the same number of columns as `b`. Each row of the product is the list of dot products of the corresponding row of `a` and the columns of `b`:

```
(defund rmat* (a b)
  (if (consp a)
      (cons (rdot-list (car a) (transpose-mat b))
            (rmat* (cdr a) b))
    ()))
(defthm rmatp-rmat*
  (implies (and (rmatp a m n) (rmatp b n p) (posp m) (posp n) (posp p))
           (rmatp (rmat* a b) m p)))
(defthmd rmat*-entry
  (implies (and (posp m) (posp n) (posp p) (rmatp a m n) (rmatp b n p)
                (natp i) (< i m) (natp j) (< j p))
           (equal (entry i j (rmat* a b))
                  (rdot (row i a) (col j b)))))
```

The formula for the transpose of a product is an immediate consequence of `transpose-rmat-entry`, `rmat*-entry`, and `rmat-entry-diff-lemma`:

```
(defthmd transpose-rmat*
  (implies (and (posp m) (posp n) (posp p) (rmatp a m n) (rmatp b n p))
           (equal (transpose-mat (rmat* a b))
                  (rmat* (transpose-mat b) (transpose-mat a)))))
```

For `i` $<$ `n`, row `i` of the `n`$\times$`n` *identity matrix* is the *unit vector* `(runit i n)`, the rlist of length `n` with 1 at index `i` and 0 elsewhere:

```
(defun runit (i n)
  (if (zp n) ()
    (if (zp i) (cons (r1) (rlistn0 (1- n)))
      (cons (r0) (runit (1- i) (1- n))))))
(defun id-rmat-aux (i n)
  (if (and (natp i) (natp n) (< i n))
```

```
      (cons (runit i n) (id-rmat-aux (1+ i) n))
    ()))
(defund id-rmat (n) (id-rmat-aux 0 n))
```

The entries of the identity matrix are given by the *Kronecker delta* function:

```
(defun rdelta (i j) (if (= i j) (r1) (r0)))
(defthmd entry-id-rmat
  (implies (and (natp n) (natp i) (natp j) (< i n) (< j n))
           (equal (entry i j (id-rmat n)) (rdelta i j))))
```

It follows that the identity matrix is its own transpose, which in turn implies its defining properties:

```
(defthmd transpose-id-rmat
  (implies (natp n) (equal (transpose-mat (id-rmat n)) (id-rmat n))))
(defthmd id-rmat-right
  (implies (and (posp m) (posp n) (rmatp a m n))
           (equal (rmat* a (id-rmat n)) a)))
(defthmd id-rmat-left
  (implies (and (posp m) (posp n) (rmatp a m n))
           (equal (rmat* (id-rmat m) a) a)))
```

To prove associativity of multiplication, let `a`, `b`, and `c` be matrices of dimensions `m`×`n`, `n`×`p`, and `p`×`q`, respectively, so that both products `(rmat a (rmat* b c))` and `(rmat* (rmat* a b) c)` are `m`×`q` matrices. It will suffice to show that corresponding entries agree:

```
(entry i j (rmat* a (rmat* b c))) = (entry i j (rmat* (rmat* a b) c)).
```
(1)

The usual informal proof proceeds by expanding the matrix products as well as the resulting dot products. In standard notation (e.g., writing a_{ir} for `(entry i r a)`), the resulting goal is

$$\sum_{r=0}^{n-1}\sum_{s=0}^{p-1} a_{ir}b_{rs}c_{sj} = \sum_{s=0}^{p-1}\sum_{r=0}^{n-1} a_{ir}b_{rs}c_{sj}.$$

The proof is completed by simply observing that the sum on the right is a rearrangement of the three-way products that appear in the sum on the left. Our objective is a formal proof that captures the intuition underlying this observation.

We shall show that these products are the entries of the `n`×`p` matrix `(rmat12 a b c i j)`, defined as follows:

```
(defun rlist-mul-list (x l)
  (if (consp l)
      (cons (rlist-mul x (car l))
            (rlist-mul-list x (cdr l)))
    ()))
(defun rlist-scalar-mul-list (x l)
  (if (consp l)
      (cons (rlist-scalar-mul (car x) (car l))
            (rlist-scalar-mul-list (cdr x) (cdr l)))
    ()))
(defund rmat12 (a b c i j)
  (rlist-scalar-mul-list (row i a) (rlist-mul-list (col j c) b)))
```

To compute the entries of this matrix, first we compute its `r`th row:

```
(nth r (rmat12 a b c i j))
  = (rlist-scalar-mul (nth r (row i a)) (nth r (rlist-mul-list (col j c) b)))
  = (rlist-scalar-mul (entry i r a) (rlist-mul (col j c) (nth r b)))
```

Now the s`th entry of the r`th row:

```
(entry r s (rmat12 a b c i j))
  = (nth s (nth r (rmat12 a b c i j)))
  = (nth s (rlist-scalar-mul (entry i r a) (rlist-mul (col j c) (nth r b))))
  = (entry i r a) * (nth s (rlist-mul (col j c) (nth r b)))
  = (entry i r a) * ((nth s (col j c)) * (nth s (nth r b)))
  = (entry i r a) * ((entry s j c) * (entry r s b))
  = (entry i r a) * (entry r s b) * (entry s j c)
```

Next we compute `(rmat-sum (rmat12 a b c i j))`. As a first step, it is easily shown by induction that if `x` is an rlist of length `n` and `l` is a matrix with `n` rows, then

```
(rmat-sum (rlist-scalar-mul-list x l)) = (rdot x (rlist-sum-list l)).
```

We apply this result to the definition of `rmat-sum`, substituting `(row i a)` for `x` and `(rlist-mul-list (col j c) b)` for `l`. This yields the following expression for `rmat-sum (rmat12 a b c i j)`:

```
(rdot (row i a) (rlist-sum-list (rlist-mul-list (col j c) b))).
```

Note that `(rlist-sum-list (rlist-mul-list (col j c) b))` and `(col j (rmat* b c))` are both rlists of length `n`. To prove equality, it suffices to show that corresponding members are equal:

```
(nth k (rlist-sum-list (rlist-mul-list (col j c) b)))
  = (rlist-sum (nth k (rlist-mul-list (col j c) b)))
  = (rlist-sum (rlist-mul (col j c) (nth k b)))
  = (rdot (col j c) (nth k b))
  = (rdot (col j c) (row k b))
  = (rdot (row k b) (col j c))
  = (entry k j (rmat* b c))
  = (nth k (col j (rmat* b c)))
```

Thus, `(rlist-sum-list (rlist-mul-list (col j c) b)) = (col j (rmat* b c))`. It follows that

```
(rmat-sum (rmat12 a b c i j)) = (rdot (row i a) (col j (rmat* b c)))
                              = (entry i j (rmat* a (rmat* b c))):
```

The `p`×`n` matrix corresponding to the right side of Equation (1) is similarly defined:

```
(defund rmat21 (a b c i j)
  (rlist-scalar-mul-list (col j c)
                         (rlist-mul-list (row i a) (transpose-mat b))))
```

Minor variations in the above derivations yield an expression for the entries of this matrix,

```
(entry r s (rmat21 a b c i j)) = (r* (entry i s a) (r* (entry s r b) (entry r j c))),
```

and the sum of these entries:

```
(rmat-sum (rmat21 a b c i j)) = (entry i j (rmat* (rmat* a b) c)).
```

Thus, `(entry r s (rmat21 a b c i j)) = (entry s r (rmat12 a b c i j))`, and hence

```
(transpose-mat (rmat12 a b c i j)) = (rmat21 a b c i j).
```

Finally, Equation (1) follows from `sum-rmat-transpose`, and associativity holds:

```
(defthmd rmat*-assoc
  (implies (and (rmatp a m n) (rmatp b n p) (rmatp c p q)
                (posp m) (posp n) (posp p) (posp q))
           (equal (rmat* a (rmat* b c))
                  (rmat* (rmat* a b) c))))
```

4 Determinants

In `rdet.lisp`, we formalize the classical definition of the *determinant* of an `n`×`n` matrix over the ring `R`, based on the symmetric group `(sym n)` as defined in `books/projects/groups/symmetric.lisp` and documented in [13]. The elements of this group are the members of the list `(slist n)` of permutations of the list `(ninit n)` = `(0 1 ... n-1)`. Such a permutation `p` may be viewed as a bijection of `(ninit n)` that maps an index `j` to `(nth j p)`. The composition of permutations `p` and `q` is computed by the group operation, `(comp-perm p q n)`. Note that `(ninit n)` itself is the group identity.

A *transposition* is a permution, denoted by `(transpose i j n)`, that simply interchanges two distinct indices `i` and `j`. Every permutation may be represented as a composition of a list of transpositions, and while neither this list nor its length is unique, its length is either always even or always odd for a given permutation `p`; `p` is said to be *even* or *odd* accordingly.

A permutation `p` is applied to an arbitrary list `l` of length `n` by the following function:

```
(defun permute (l p)
  (if (consp p)
      (cons (nth (car p) l) (permute l (cdr p)))
    ()))
```

A critical property of `permute` pertains to a product of permutations:

```
(defthm permute-comp-perm
  (implies (and (true-listp l) (consp l) (in x (sym (len l))) (in y (sym (len l))))
           (equal (permute (permute l x) y)
                  (permute l (comp-perm x y (len l))))))
```

Each permutation `p` in `(sym n)` contributes a term `(rdet-term a p n)` to the determinant of an `n`×`n` matrix `a`, computed as follows:

(1) For each `i` < `n`, select the entry of `(row i a)` in column `(nth i p)`;

(2) Compute the product of these `n` entries;

(3) Negate the product if `p` is an odd permutation.

```
(defun rdet-prod (a p n)
  (if (zp n)
      (r1)
    (r* (rdet-prod a p (1- n))
        (entry (1- n) (nth (1- n) p) a))))
(defund rdet-term (a p n)
  (if (even-perm-p p n)
      (rdet-prod a p n)
    (r- (rdet-prod a p n))))
```

The determinant of `a` is the the sum over `(slist n)` of these signed products:

```
(defun rdet-sum (a l n)
  (if (consp l)
      (r+ (rdet-term a (car l) n) (rdet-sum a (cdr l) n))
    (r0)))
(defund rdet (a n) (rdet-sum a (slist n) n))
```

4.1 Properties

To compute the determinant of the identity matrix, note that if p is any permutation other than the identity (ninit n), we can find i < n such that (nth i p) ≠ i, and hence (entry i (nth i p) (id-rmat n)) = 0, which implies (rdet-term (id-rmat n) p n) = 0. On the other hand, (nth i (ninit n)) = i for all i, which implies (rdet-term (id-rmat n) (ninit n) n) = 1. Thus,

```
(defthm rdet-id-rmat (implies (posp n) (equal (rdet (id-rmat n) n) (r1))))
```

The determinant is invariant under transpose-mat. This follows from the observation that the term contributed to the determinant of the transpose of a by a permutation p is the same as the term contributed to the determinant of a by the inverse of p:

```
(defthmd rdet-transpose
  (implies (and (posp n) (rmatp a n n))
           (equal (rdet (transpose-mat a) n) (rdet a n))))
```

If every entry of the kth row of a is 0, then for all p, the kth factor of (rdet-prod a p n) is 0, and it follows that the determinant of a is 0:

```
(defthmd rdet-row-0
  (implies (and (rmatp a n n) (posp n) (natp k) (< k n) (= (nth k a) (rlistn0 n)))
           (equal (rdet a n) (r0))))
```

Furthermore, the determinant is *alternating*, i.e., if two rows of a are equal, then its determinant is 0. To prove this, suppose rows i and j are equal, where i ≠ j. Given a permutation p, let p' = (comp-perm p (transpose i j n) n). The factors of (rdet-prod a p' n) are the same as those of (rdet-prod a p n). But p and p' have opposite parities, and therefore (rdet-term a p' n) is the negative of (rdet-term a p n). Consequently, the sum of terms contributed by the odd permutations is the negative of the sum of terms contributed by the even permutations, and we have

```
(defthmd rdet-alternating
  (implies (and (rmatp a n n) (posp n)
                (natp i) (< i n) (natp j) (< j n) (not (= i j))
                (= (row i a) (row j a)))
           (equal (rdet a n) (r0))))
```

The determinant is also *n-linear*, i.e., linear as a function of each row. This property is specified in terms of the replace-row operation. For a given row i and permutation p, the term contributed by p to the determinant of (replace-row a i x) is a linear function of x:

```
(defthm rdet-term-replace-row
  (implies (and (rmatp a n n) (posp n) (member p (slist n))
                (rlistnp x n) (rlistnp y n) (rp c)
                (natp i) (< i n))
           (let ((a1 (replace-row a i x))
                 (a2 (replace-row a i y))
                 (a3 (replace-row a i (rlist-add (rlist-scalar-mul c x) y))))
             (equal (rdet-term a3 p n)
                    (r+ (r* c (rdet-term a1 p n)) (rdet-term a2 p n))))))
```

The desired result follows by summing over all permutations:

```
(defthm rdet-n-linear
  (implies (and (rmatp a n n) (posp n) (natp i) (< i n)
                (rlistnp x n) (rlistnp y n) (rp c))
           (equal (rdet (replace-row a i (rlist-add (rlist-scalar-mul c x) y)) n)
                  (r+ (r* c (rdet (replace-row a i x) n))
                      (rdet (replace-row a i y) n)))))
```

4.2 Uniqueness

We shall show that rdet is the unique n-linear alternating function on n×n matrices that satisfies (rdet (id-rmat n) n) = 1. To that end, we introduce a constrained function rdet0 as follows:

```
(encapsulate (((rdet0 * *) => *))
  (local (defun rdet0 (a n) (rdet a n)))
  (defthm rp-rdet0
    (implies (and (rmatp a n n) (posp n))
             (rp (rdet0 a n))))
  (defthmd rdet0-n-linear
    (implies (and (rmatp a n n) (posp n) (natp i) (< i n)
                  (rlistnp x n) (rlistnp y n) (rp c))
             (equal (rdet0 (replace-row a i (rlist-add (rlist-scalar-mul c x) y)) n)
                    (r+ (r* c (rdet0 (replace-row a i x) n))
                        (rdet0 (replace-row a i y) n)))))
  (defthmd rdet0-adjacent-equal
    (implies (and (rmatp a n n) (posp n)
                  (natp i) (< i (1- n)) (= (row i a) (row (1+ i) a)))
             (equal (rdet0 a n) (r0)))))
```

Our main objective is to prove that

```
(rdet0 a n) = (r* (rdet a n) (rdet0 (id-rmat n))).
```
(2)

If we then prove that a given function (f a n) satisfies the constraints on rdet0, then we may conclude by functional instantiation that (f a n) = (r* (rdet a n) (f (id-rmat n) n)). From this it will follow that if f has the additional property (f (id-rmat n) n) = 1, then (f a n) = (rdet a n).

Note that instead of assuming that rdet0 is alternating, we have imposed the weaker constraint rdet0-adjacent-equal, which says that the value is 0 if two *adjacent* rows are equal. This relaxes the proof obligations for functional instantiation, which will be critical for the proof of correctness of cofactor expansion (Section 5). However, it is a consequence of the above constraints that rdet0 is alternating. To establish this, we first show by a sequence of applications of rdet0-n-linear and rdet0-adjacent-equal that transposing two adjacent rows negates the value of rdet0. It is also easily shown that an arbitrary transposition may be expressed as a composition of an odd number of transpositions of adjacent rows, and it follows that the value is negated by transposing any two rows:

```
(defthmd rdet0-permute-transpose
  (implies (and (rmatp a n n) (posp n)
                (natp i) (natp j) (< i j) (< j n))
           (equal (rdet0 (permute a (transpose i j n)) n)
                  (r- (rdet0 a n)))))
```

Since every permutation is a product of transpositions, this yields the following generalization:

```
(defthmd rdet0-permute-rows
  (implies (and (rmatp a n n) (posp n) (in p (sym n)))
           (equal (rdet0 (permute a p) n)
                  (if (even-perm-p p n)
                      (rdet0 a n)
                    (r- (rdet0 a n))))))
```

Now suppose (row i a) = (row j a), where $0 \leq$ i $<$ j $<$ n. By rdet0-adjacent-equal, we may also assume i + 1 $<$ j. Let a' = (permute (transpose (1+ i) j n) a). Then

```
(nth (1+ i) a') = (nth j a) = (nth i a) = (nth i a').
```

By `rdet0-adjacent-equal`, `(rdet0 a')` = 0, and by `rdet0-permute-transpose`,

```
(rdet0 a n) = (r- (rdet0 a' n) = (r- 0) = 0.
```

Thus, `rdet0` is an alternating function:

```
(defthmd rdet0-alternating
  (implies (and (rmatp a n n) (posp n) (natp i) (natp j) (< i n) (< j n)
                (not (= i j)) (= (row i a) (row j a)))
           (equal (rdet0 a n) (r0))))
```

Our proof of Equation (2) involves arbitrary lists of length `k` $\leq$ `n` of natural numbers less than `n`, which we call `k`-*tuples*. We begin with the following definitions:

- `(tuplep x k n)` is a predicate that recognizes a `k`-tuple;
- `(extend-tuple x n)` returns the list of `n` (`k`+1)-tuples constructed from a given `k`-tuple `x` by appending each natural number less than `n`;
- `(extend-tuples l n)` returns the list of all (`k`+1)-tuples constructed in this way from the members of a list `l` of `k`-tuples.

The list of all `k`-tuples is defined recursively:

```
(defun all-tuples (k n)
  (if (zp k)
      (list ())
    (extend-tuples (all-tuples (1- k) n) n)))
```

Let `a` be a fixed `n`×`n` matrix. We associate a value `(reval-tuple x k a n)` with each `k`-tuple `x` as follows. First we construct an rlist of length `k`, `(extract-entries x a)`, the `j`th member of which is `(entry j (nth j x) a)`:

```
(defun extract-entries (x a)
  (if (consp x)
      (cons (nth (car x) (car a))
            (extract-entries (cdr x) (cdr a)))
    ()))
```

We define `(runits x n)` to be the list of unit vectors corresponding to the members of `x`:

```
(defun runits (x n)
  (if (consp x)
      (cons (runit (car x) n) (runits (cdr x) n))
    ()))
```

The value `(reval-tuple x k a n)` is the product of the members of `(extract-entries x a)` together with the value of `rdet0` applied to the matrix derived from `a` by replacing its first `k` rows with `(runits x n)`:

```
(defun reval-tuple (x k a n)
  (r* (rlist-prod (extract-entries x a))
      (rdet0 (append (runits x n) (nthcdr k a)) n)))
```

We also define the sum of the values of `(reval-tuple x k a n)` as `x` ranges over a list `l` of `k`-tuples:

```
(defun rsum-tuples (l k a n)
  (if (consp l)
      (r+ (reval-tuple (car l) k a n) (rsum-tuples (cdr l) k a n))
    (r0)))
```

We would like to compute `(rsum-tuples (all-tuples k n) k a n)`. Since the only member of `(all-tuples 0 n)` is `NIL`, the case k = 0 is trivial:

```
(rsum-tuples (all-tuples 0 n) 0 a n) = (reval-tuple () 0 a n) = (rdet0 a n).    (3)
```

For the case k = n, we observe that `(nthcdr n a)` = `NIL` and that if the members of `x` are not distinct, then the matrix `(runits x n)` has two equal rows and by `rdet0-alternating`, `(rdet0 (runits x n) n)` = 0. Thus, in the computation of `(rsum-tuples (all-tuples n n) n a n)`, we need only consider the n-tuples that are permutations. If `p` is in `(sym n)`, then by `rdet0-permute-rows`,

```
(rdet0 (runits p n) n)
  = (rdet0 (permute (id-rmat n) p) n)
  = (if (even-perm-p n n) (rdet0 (id-rmat n) n) (r- (rdet0 (id-rmat n) n)))
```

and `(extract-entries p a)` = `(rdet-prod a p n)`. Consequently,

```
(reval-tuple p n a n) = (r* (rdet-term a p n) (rdet0 (id-rmat n) n)).
```

Summing over `(slist n)`, we have

```
(rsum-tuples (all-tuples n n) n a n) = (r* (rdet a n) (rdet0 (id-rmat n) n)).    (4)
```

For $0 \leq k < n$ and `(tuplep x k n)`, repeated application of `rdet0-n-linear` yields

```
(rsum-tuples (extend-tuple x) (1+ k) a n) = (reval-tuple x k a n).
```

Summing over `(all-tuples k n)`, we have the recurrence formula

```
(rsum-tuples (all-tuples (1+ k) n) (1+ k) a n) = (rsum-tuples (all-tuples k n) k a n).
```

By induction, `(rsum-tuples (all-tuples k n) k a n)` is independent of `k`. In particular,

```
(rsum-tuples (all-tuples n n) n a n) = (rsum-tuples (all-tuples 0 n) 0 a n).
```

Equation (2) follows from this result together with Equations (3) and (4):

```
(defthmd rdet-unique
  (implies (rmatp a n n)
           (equal (rdet0 a n)
                  (r* (rdet a n) (rdet0 (id-rmat n) n)))))
```

4.3 Multiplicativity

If we had further constrained the function `rdet0` to satisfy `(rdet0 (id-rmat n) n)` = 1, then we could have replaced the conclusion of `rdet-unique` with the simpler equation `(rdet0 a n)` = `(rdet a n)`. One reason behind our weaker specification is that it allows us to prove the multiplicativity property, `(rdet (rmat* a b) n)` = `(r* (rdet a n) (rdet b n)`, by functional instantiation. We define

```
(defun rdet-rmat* (a b n) (rdet (rmat* a b) n))
```

Our goal is the functional instance of `rdet-unique` derived by substituting

```
(lambda (a n) (rdet-rmat* a b n))
```

for `rdet0`. This requires that we prove the analogs of the two nontrivial constraints on `rdet0`. The first is a consequence of `rdet-n-linear` and the definitions of `rmat*`, `rdot-list`, and `rlist-scalar-mul`:

```
(defthmd rdet-rmat*-n-linear
  (implies (and (rmatp a n n) (rmatp b n n) (posp n) (natp k) (< k n)
                (rlistnp x n) (rlistnp y n) (rp c))
           (equal (rdet-rmat* (replace-row a k (rlist-add (rlist-scalar-mul c x) y))
                              b n)
                  (r+ (r* c (rdet-rmat* (replace-row a k x) b n))
                      (rdet-rmat* (replace-row a k y) b n)))))
```

The second follows from `rdet-alternating` and the observation that if `(row k a) = (row (1+ k) a)`, then `(row k (rmat* a b)) = (row (1+ k) (rmat* a b))`:

```
(defthmd rdet-rmat*-adjacent-equal
  (implies (and (rmatp a n n) (rmatp b n n) (posp n)
                (natp k) (< k (1- n)) (= (row k a) (row (1+ k) a)))
           (equal (rdet-rmat* a b n) (r0))))
```

Functional instantiation of `rdet-unique` yields

```
(rdet-rmat* a b n) = (r* (rdet a n) (rdet-rmat* (id-rmat n) b n)).
```

Expanding `rdet-rmat*` and applying `id-rmat-left`, we have

```
(defthmd rdet-multiplicative
  (implies (and (rmatp a n n) (rmatp b n n) (posp n))
           (equal (rdet (rmat* a b) n)
                  (r* (rdet a n) (rdet b n)))))
```

5 Cofactors

Given an `n`×`n` matrix `a`, we define the `(n-1)`×`(n-1)` submatrix `(minor i j a)` to be the result of deleting the `i`th row and the `j`th column of `a`:

```
(defun delete-row (k a)
  (if (zp k) (cdr a)
    (cons (car a) (delete-row (1- k) (cdr a)))))
(defund delete-col (k a) (transpose-mat (delete-row k (transpose-mat a))))
(defund minor (i j a) (delete-col j (delete-row i a)))
```

Its entries may be computed as follows:

```
(defthmd entry-rmat-minor
  (implies (and (rmatp a n n) (natp n) (> n 1) (natp i) (natp j) (< i n) (< j n)
                (natp r) (natp s) (< r (1- n)) (< s (1- n)))
           (equal (entry r s (minor i j a))
                  (entry (if (>= r i) (1+ r) r) (if (>= s j) (1+ s) s) a))))
```

The *cofactor* of an entry of `a` is the determinant of its minor with an attached sign determined by the parity of the sum of its indices:

```
(defund rdet-cofactor (i j a n)
  (if (evenp (+ i j))
      (rdet (minor i j a) (1- n))
    (r- (rdet (minor i j a) (1- n)))))
```

5.1 Cofactor Expansion

The cofactor expansion of the determinant of a by a column is computed by multiplying each entry of the column by its cofactor and summing the products:

```
(defun expand-rdet-col-aux (a i j n)
  (if (zp i) (r0)
    (r+ (r* (entry (1- i) j a) (rdet-cofactor (1- i) j a n))
        (expand-rdet-col-aux a (1- i) j n))))
(defund expand-rdet-col (a j n) (expand-rdet-col-aux a n j n))
```

Cofactor expansion by a row is similarly defined:

```
(defun expand-rdet-row-aux (a i j n)
  (if (zp j) (r0)
    (r+ (r* (entry i (1- j) a) (rdet-cofactor i (1- j) a n))
        (expand-rdet-row-aux a i (1- j) n))))
(defund expand-rdet-row (a i n) (expand-rdet-row-aux a i n n))
```

It follows from `entry-rmat-minor` and `transpose-rmat-entry` that

```
(transpose-mat (minor i j a)) = (minor j i (transpose-mat a)),
```

which, in combination with `rdet-transpose`, implies

```
(rdet-cofactor j i (transpose-mat a) n) = (rdet-cofactor i j a n).
```

Consequently, cofactor expansion by column `i` is equivalent to expansion of the transpose by row `i`:

```
(defthmd expand-rdet-row-transpose
  (implies (and (rmatp a n n) (natp n) (> n 1) (natp i) (< i n))
           (equal (expand-rdet-row (transpose-mat a) i n)
                  (expand-rdet-col a i n))))
```

We shall prove, by functional instantiation of `rdet-unique`, that the result of cofactor expansion by a column has the same value as the determinant, and it will follow that the same is true for expansion by a row. Once again, this requires proving analogs of the constraints on `rdet0`.

It is clear that replacing row `i` of `a` does not alter `(rdet-cofactor i j a b)`. On the other hand, for $\mathtt{k} \neq \mathtt{i}$, `(rdet-cofactor i j a n)` is a linear function of `(row k a)`:

```
(defthmd rdet-cofactor-n-linear
  (implies (and (rmatp a n n) (natp n) (> n 1) (natp i) (< i n) (natp j) (< j n)
                (natp k) (< k n) (not (= k i)) (rlistnp x n) (rlistnp y n) (rp c))
           (equal (rdet-cofactor
                     i j (replace-row a k (rlist-add (rlist-scalar-mul c x) y)) n)
                  (r+ (r* c (rdet-cofactor i j (replace-row a k x) n))
                      (rdet-cofactor i j (replace-row a k y) n)))))
```

It follows that cofactor expansion by column `j` is `n`-linear:

```
(defthmd expand-rdet-col-n-linear
  (implies (and (rmatp a n n) (natp n) (> n 1) (natp j) (< j n)
                (natp k) (< k n) (rlistnp x n) (rlistnp y n) (rp c))
           (equal (expand-rdet-col
                    (replace-row a k (rlist-add (rlist-scalar-mul c x) y)) j n)
                  (r+ (r* c (expand-rdet-col (replace-row a k x) j n))
                      (expand-rdet-col (replace-row a k y) j n)))))
```

Now suppose adjacent rows k and k + 1 are equal. Then for any index i other than k or k + 1, `(minor i j a)` has two equal adjacent rows, and therefore `(rdet-cofactor i j a n)` = 0. Meanwhile,

```
(minor k j) = (minor (1+ k) j)
```

and

```
(entry k j a) = (entry (1+ k) j a),
```

but k + j and (k + 1) + j have opposite parities, and hence

```
(rdet-cofactor k j a n) + (rdet-cofactor (1+ k) j a n) = 0.
```

Therefore, `(expand-rdet-col a j n)` = 0:

```
(defthmd expand-rdet-col-adjacent-equal
  (implies (and (rmatp a n n) (> n 1) (natp j) (< j n)
                (natp k) (< k (1- n)) (= (row k a) (row (1+ k) a)))
           (equal (expand-rdet-col a j n) (r0))))
```

Thus, the constraints on `rdet0` are satisfied, and by functional instantiation of `rdet-unique`, we have the following:

```
(defthmd expand-rdet-col-val
  (implies (and (rmatp a n n) (posp n) (> n 1) (natp k) (< k n))
           (equal (expand-rdet-col a k n)
                  (r* (rdet a n) (expand-rdet-col (id-rmat n) k n)))))
```

It remains to show that `(expand-rdet-col (id-rmat n) k n)` = 1. By `row-rmat-minor` (see `rdet.lisp`), we have the following expression for a row of `(minor i j (id-rmat n))`:

```
(defthmd nth-minor-id-rmat
  (implies (and (natp n) (> n 1) (natp i) (< i n) (natp j) (< j n)
                (natp r) (< r (1- n)))
           (equal (nth r (minor i j (id-rmat n)))
                  (delete-nth j (runit (if (< r i) r (1+ r)) n)))))
```

The following is a consequence of the definitions of `runit` and `delete-nth`:

```
(defthmd delete-nth-runit
  (implies (and (posp n) (natp j) (< j n) (natp k) (< k n))
           (equal (delete-nth j (runit k n))
                  (if (< j k) (runit (1- k) (1- n))
                    (if (> j k) (runit k (1- n))
                      (rlistn0 (1- n)))))))
```

Consequently, if i ≠ j, then we find a zero row of `(minor i j (id-rmat n))`, and by `rdet-row-0`, its determinant is 0. On the other hand, `(minor j j (id-rmat n))` = `(id-rmat (1- n))` and the corresponding cofactor is 1, as is the cofactor expansion:

```
(defthmd expand-rdet-col-id-rmat
  (implies (and (rmatp a n n) (natp n) (> n 1) (natp j) (< j n))
           (equal (expand-rdet-col (id-rmat n) j n) (r1))))
```

Combining this with `expand-rdet-col-val`, we have the correctness theorem for column expansion:

```
(defthmd expand-rdet-col-rdet
  (implies (and (rmatp a n n) (posp n) (> n 1) (natp k) (< k n))
           (equal (expand-rdet-col a k n) (rdet a n))))
```

It follows from `rdet-transpose`, `expand-rdet-row-transpose`, and `transpose-rmat-2` that the same holds for row expansion:

```
(defthmd expand-rdet-row-rdet
  (implies (and (rmatp a n n) (posp n) (> n 1) (natp k) (< k n))
           (equal (expand-rdet-row a k n) (rdet a n))))
```

As a consequence of `expand-rdet-row-rdet`, we have a recursive version of `rdet`, based on cofactor expansion with respect to row 0:

```
(mutual-recursion
  (defund rdet-rec-cofactor (j a n)
    (if (zp n) ()
      (if (evenp j) (rdet-rec (minor 0 j a) (1- n))
        (r- (rdet-rec (minor 0 j a) (1- n))))))
  (defun expand-rdet-rec-aux (a j n)
    (if (zp j) (r0)
      (r+ (r* (entry 0 (1- j) a) (rdet-rec-cofactor (1- j) a n))
          (expand-rdet-rec-aux a (1- j) n))))
  (defund expand-rdet-rec (a n) (expand-rdet-rec-aux a n n))
  (defun rdet-rec (a n)
    (if (zp n) (r0)
      (if (= n 1) (entry 0 0 a)
        (expand-rdet-rec a n)))))
```

The equivalence follows from `expand-rdet-row-rdet` by induction (see `rdet.lisp` for details):

```
(defthmd rdet-rec-rdet
  (implies (and (rmatp a n n) (posp n))
           (equal (rdet-rec a n) (rdet a n))))
```

5.2 Classical Adjoint

We shall define the *cofactor matrix* of an `n`×`n` matrix `a` to be the `n`×`n` matrix with entries

```
(entry i j (cofactor-rmat a b)) = (rdet-cofactor i j a n).
```

To define this matrix, we first define a function that computes its `i`th row:

```
(defun cofactor-rmat-row-aux (i j a n)
  (if (and (natp n) (> n 1) (natp j) (< j n))
      (cons (rdet-cofactor i j a n) (cofactor-rmat-row-aux i (1+ j) a n))
    ()))
(defund cofactor-rmat-row (i a n) (cofactor-rmat-row-aux i 0 a n))

(defun cofactor-rmat-aux (i a n)
  (if (and (natp n) (natp i) (< i n))
      (cons (cofactor-rmat-row i a n) (cofactor-rmat-aux (1+ i) a n))
    ()))
(defund cofactor-rmat (a n) (cofactor-rmat-aux 0 a n))
```

The *classical adjoint* of a is the transpose of its cofactor matrix:

```
(defund adjoint-rmat (a n) (transpose-mat (cofactor-rmat a n)))
```

The following is an equivalent formulation:

```
(defthmd cofactor-rmat-transpose
  (implies (and (rmatp a n n) (natp n) (> n 1))
           (equal (cofactor-rmat (transpose-mat a) n)
                  (adjoint-rmat a n))))
```

Note that the dot product of `(row i a)` with `(cofactor-rmat-row i a n)` is a rearrangement of the sum `(expand-rdet-row a i n)`:

```
(defthmd rdot-cofactor-rmat-row-expand-rdet-row
  (implies (and (rmatp a n n) (natp n) (> n 1) (natp i) (< i n))
           (equal (rdot (row i a) (cofactor-rmat-row i a n))
                  (expand-rdet-row a i n))))
```

Combining this with `expand-rdet-row-rdet`, we have the following expression for the determinant:

```
(defthmd rdot-cofactor-rmat-row-rdet
  (implies (and (rmatp a n n) (natp n) (> n 1) (natp i) (< i n))
           (equal (rdot (row i a) (cofactor-rmat-row i a n))
                  (rdet a n))))
```

Next we consider the result of substituting `(replace-row a i (row k a))` for a in `rdot-cofactor-rmat-row-rdet`, where $k \neq i$. Since this matrix has two identical rows, its determinant is 0, and we have

```
(defthmd rdot-cofactor-rmat-row-rdet-0
  (implies (and (rmatp a n n) (natp n) (> n 1) (natp i) (< i n)
                (natp k) (< k n) (not (= k i)))
           (equal (rdot (row k a) (cofactor-rmat-row i a n))
                  (r0))))
```

Thus, we have the following for general k:

```
(defthmd rdot-cofactor-rmat-row-rdelta
  (implies (and (rmatp a n n) (natp n) (> n 1) (natp i) (< i n) (natp k) (< k n))
           (equal (rdot (row k a) (cofactor-rmat-row i a n))
                  (r* (rdelta i k) (rdet a n)))))
```

Since `(cofactor-rmat-row i a n)` = `(col i (adjoint-mat a n))`, this yields an expression for the n×n matrix product of a and its adjoint:

```
(defthmd rmat*-adjoint-rmat
  (implies (and (rmatp a n n) (natp n) (> n 1))
           (equal (rmat* a (adjoint-rmat a n))
                  (rmat-scalar-mul (rdet a n) (id-rmat n)))))
```

In Part II, where we consider matrices with entries ranging over a field, we shall use this last equation in deriving a criterion for the existence of a multiplicative inverse of a matrix. We shall also apply the results of this subsection to a proof of Cramer's Rule for solving a system of n linear equations in n unknowns.

References

[1] William Brown (1993): *Matrices over Commutative Rings*. M. Dekker.

[2] Ruben Gamboa, John Cowles & Jeff Van Baalen (2003): *Using ACL2 Arrays to Formalize Matrix Algebra*. In: *ACL2 2003: 4th International Workshop on the ACL2 Theorem Prover and its Applications*, Boulder, Colorado.

[3] Joe Hendrix (2003): *Matrices in ACL2*. In: *ACL2 2003: 4th International Workshop on the ACL2 Theorem Prover and its Applications*, Boulder, Colorado.

[4] Kenneth Hoffman & Ray Kunze (1961): *Linear Algebra*. Allyn Prentice-Hall.

[5] Bernard Kolman (1977): *Elementary Linear Algebra*, 2nd edition. MacMillan.

[6] Jin Ho Kwak & Sungpyo Kong (1997): *Linear Algebra*. Birkhäuser. doi:10.1007/978-1-4757-1200-1.

[7] Carl Kwan & Warren Hunt (2024): *Automatic Verification of Right-greedy Numerical Linear Algebra Algorithms*. In: *Proceedings of the 24th Conference on Formal Methods in Computer-Aided Design (FMCAD 2024)*, doi:10.34727/2024/isbn.978-3-85448-065-5.

[8] Carl Kwan & Warren Hunt (2024): *Formalizing the Cholesky Factorization Theorem*. In: *Proceedings for the Fifteenth Conference on Interactive Theorem Proving (ITP 2024)*, doi:10.4230/LIPIcs.ITP.2024.25.

[9] *Maths in Lean: Linear Algebra*. Available at `https://leanprover-community.github.io/theories/linear_algebra.html`.

[10] Steven Roman (2005): *Advanced Linear Algebra*, 2nd edition. Springer, doi:10.1007/978-1-4757-2178-2.

[11] 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.

[12] 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, doi:10.4204/EPTCS.393.4.

[13] David M. Russinoff (2023): *A Formalization of Finite Froup Theory: Part III*. In: *ACL2 2023: 18th International Workshop on the ACL2 Theorem Prover and its Applications*, Austin, Texas, doi:10.4204/EPTCS.393.5.

[14] David M. Russinoff (2025): *A Formalization of Elementary Linear Algebra: Part II*. In: *ACL2 2025: 19th International Workshop on the ACL2 Theorem Prover and its Applications*, Austin, Texas.

[15] ZhengPu Shi & Gang Chen (2022): *Integration of Multiple Formal Matrix Models in Coq*. In Wei Dong & Jean-Pierre Talpin, editors: *Dependable Software Engineering Theories, Tools, and Applications*, Springer Nature Switzerland, doi:10.1007/978-3-031-21213-0_11.

[16] ZhengPu Shi & Gang Chen (2024): *Formal Verification of Executable Matrix Inversion via Adjoint Matrix and Gaussian Elimination*. In: *Proceedings of the 26th International Symposium on Principles and Practice of Declarative Programming*, doi:10.1145/3678232.3678242.

[17] Zhiping Shi, Yan Zhang, Zhenke Liu, Ximan Kank, Yong Guan, Jie Zhang & Xiaoyu Song (2014): *Formalization of matrix theory in Hol4*. In: *Advances in Mechanical Engineering*, 6, doi:10.1155/2014/195276.

[18] Christian Sternagel & Rene Thiemann (2010): *Executable Matrix Operations on Matrices of Arbitrary Dimensions*. In: *Archive of Formal Proofs*. Available at `https://www.isa-afp.org/entries/Matrix.html`.

A Formalization of Elementary Linear Algebra: Part II

David M. Russinoff
david@russinoff.com

This is the second installment of an exposition of an ACL2 formalization of elementary linear algebra. It extends the results of Part I, which covers the algebra of matrices over a commutative ring, but focuses on aspects of the theory that apply only to matrices over a field: elementary row reduction and its application to the computation of matrix inverses and the solution of simultaneous systems of linear equations.

1 Introduction

This is the second installment of an exposition of an ACL2 formalization of elementary linear algebra. Part I [6], which is also included in this workshop, covers the algebra of matrices over a commutative ring with unity and their determinants. In this sequel, we focus on aspects of the theory of matrices that apply only to matrices over a field, i.e., depend on the existence of a multiplicative inverse operator. These include row reduction and its application to matrix invertibility and the solution of systems of linear equations. In an anticipated Part III, all of these results will be applied to the study of abstract vector spaces and linear transformations.

The proof scripts supporting both papers reside in the same directory, `books/projects/linear/`. As described in [6], the abstract definition of a ring is formalized in the file `ring.lisp` by a set of constrained encapsulated functions: a predicate `rp` that recognizes ring elements, the binary addition and multiplication operations `r+` and `r*`, the corresponding identity constants `r0` and `r1`, and the additive inverse operator `r-`. The notion of a field is similarly defined by an encapsulation in the file `field.lisp`, in which these functions are renamed `fp`, `f+`, `f*`, etc.:

```
(encapsulate (((fp *) => *)                    ;field element recognizer
              ((f+ * *) => *) ((f* * *) => *) ;addition and multiplication
              ((f0) => *) ((f1) => *)          ;identities
              ((f- *) => *) ((f/ *) => *))     ;inverses
  (local (defun fp (x) (rationalp x)))
  (local (defun f+ (x y) (+ x y)))
  (local (defun f* (x y) (* x y)))
  (local (defun f0 () 0))
  (local (defun f1 () 1))
  (local (defun f- (x) (- x)))
  (local (defun f/ (x) (/ x)))
  ;; Closure:

  ...

  ;; Multiplicative inverse:
  (defthm fpf/
    (implies (and (fp x) (not (equal x (f0)))) (fp (f/ x))))
  (defthm f*inv
    (implies (and (fp x) (not (equal x (f0)))) (equal (f* x (f/ x)) (f1)))))
```

Gamboa and Manolios (Eds):
ACL2 Workshop 2025
EPTCS 423, 2025, pp. 19–35, doi:10.4204/EPTCS.423.2

The only other difference between the two encapsulations is the inclusion here of the multiplicative inverse `f/` along with two constraining axioms, appended to the constraints adapted from the ring axioms.

Informally we shall refer to the field `F` that is characterized by this encapsulation. When our intention is clear, the identity elements `(f0)` and `(f1)` will be abbreviated as 0 and 1. Clearly, all properties of rings hold for fields as well. Thus, all definitions and theorems that appear in `ring.lisp`, `rmat.lisp`, and `rdet.lisp` have analogs in the corresponding files `field.lisp`, `fmat.lisp`, and `fdet.lisp`. In particular, the predicate `flistnp` recognizes a list of specified length of elements of `F`, called an *flist*; `fmatp` recognizes a matrix over `F`; and `fdet` computes its determinant. In principle, all results in the latter set of files could be derived by functional instantiation from the corresponding events in the former, but we found it more expedient to reproduce the proofs, simply by selectively replacing occurrences of the character `r` with `f`. An additional file, `reduction.lisp`, contains the results reported in this paper. In describing these results, we assume the reader is familiar with Part I and is aware of the renaming convention.

In Section 2, we define the notion of a reduced row-echelon matrix and develop a procedure that converts an arbitrary matrix to reduced row-echelon form. An equivalent procedure, based on matrix multiplication, is also defined. This leads to a criterion for invertibility of a square matrix and a method for computing inverses. A second method of matrix inversion, based on determinants and the classical adjoint, is derived from the results of Part I.

Section 3 addresses the solution of systems of linear equations, mainly as an application of row reduction. We derive algorithmic tests for solvability and uniqueness of the solution, as well as a formula that computes the solution in the uniquely solvable case. For the special case of an invertible square coefficient matrix, we prove Cramer's Rule, an alternative formula based on determinants. In the general solvable case, we show that the solution set is infinite and establish a test that identifies solutions. In Part III, this will lead to a formula that generates a basis for the solution space of a homogeneous system of equations.

All of these results, which are stated and proved in the context of an abstract field, may be applied to any concrete field of interest through functional instantiation. Eventually, we plan to apply the theory to algebraic number fields. Of course, in their abstract formulation based on constrained functions, the definitions are not executable. For the immediate purposes of illustration and testing, however, all functions defined in `field.lisp`, `fmat.lisp`, `fdet.lisp`, and `reduction.lisp` have been adapted to the field of rational numbers as executable functions, which are listed in the file `rational.lisp`.

2 Row Reduction

2.1 Reduced Row-Echelon Form

A *reduced row-echelon* matrix may be characterized as follows:

(1) Every all-zero row is preceded by every nonzero row;

(2) The first nonzero entry of each nonzero row is 1, and every other entry in the same column is 0;

(3) The column of the leading 1 in the `i`th nonzero row is an increasing function of `i`.

The formalization of this definition requires several auxiliary functions. First, we define the index of the leading nonzero entry of a nonzero row `r`:

```
(defun first-nonzero (r)
  (if (consp r)
```

```
    (if (= (car r) (f0))
        (1+ (first-nonzero (cdr r)))
      0)
  ()))
```

In the following, we assume that a is an m×n matrix. Starting with row k, where $0 \leq k \leq m$, find the row of a with nonzero entry of least index, or return NIL if all rows beyond the first k are 0:

```
(defun row-with-nonzero-at-least-index (a m k)
  (if (and (natp k) (natp m) (< k m))
      (let ((i (row-with-nonzero-at-least-index a (1- m) k)))
        (if (or (flist0p (nth (1- m) a))
                (and i (<= (first-nonzero (nth i a)) (first-nonzero (nth (1- m) a)))))
            i
          (1- m)))
    ()))
```

Given $j < n$ and $k < m$, check that (entry k j a) = 1 and that all other entries in column k are 0:

```
(defun column-clear-p (a k j m)
  (if (zp m) t
    (and (= (nth j (nth (1- m) a)) (if (= (1- m) k) (f1) (f0)))
         (column-clear-p a k j (1- m)))))
```

Given $k \leq m$, check that the first k rows of a form a reduced row-echelon matrix:

```
(defun row-echelon-p-aux (a m k)
  (if (zp k) t
    (and (row-echelon-p-aux a m (1- k))
         (let ((i (row-with-nonzero-at-least-index a m (1- k))))
           (or (null i)
               (and (= i (1- k))
                    (column-clear-p a i (first-nonzero (nth i a)) m)))))))
```

Finally, check that a is a reduced row-echelon matrix:

```
(defund row-echelon-p (a) (row-echelon-p-aux a (len a) (len a)))
```

2.2 Conversion to Reduced Row-Echelon Form

We shall develop a procedure that converts an arbitrary m×n matrix a to reduced row-echelon form by a sequence of *elementary row operations* of three types:

(1) Multiply row k by a scalar c:

```
(defund ero1 (a c k) (replace-row a k (flist-scalar-mul c (nth k a))))
```

(2) Add a scalar multiple of row j to row k, where $j \neq k$:

```
(defund ero2 (a c j k)
  (replace-row a k (flist-add (flist-scalar-mul c (nth j a)) (nth k a))))
```

(3) Interchange rows j and k, where $j \neq k$:

```
(defund ero3 (a j k) (replace-row (replace-row a k (nth j a)) j (nth k a)))
```

Under the assumption that (entry k j a) = 1, the following function applies ero2 to clear all other entries in column j by adding the appropriate multiple of row k to each of the other rows:

```
(defun clear-column (a k j m)
  (if (zp m) a
    (if (= (1- m) k)
        (clear-column a k j (1- m))
      (clear-column (ero2 a (f- (nth j (nth (1- m) a))) k (1- m))
                    k j (1- m)))))
```

Assume the first k rows of a are in reduced row-echelon form, i.e., (row-echelon-p-aux a m k) = T, where $k < m$, and that i = (row-with-nonzero-at-least-index a m k) $\neq$ NIL. Let j = (first-nonzero (nth i a)). The following function performs the next step of the reduction, producing a matrix a' satisfying (row-echelon-p-aux a' m (1+ k)):

```
(defund row-reduce-step (a m k i j)
  (clear-column (ero3 (ero1 a (f/ (nth j (nth i a))) i)
                      i k)
                k j m))
```

The function row-reduce converts a to a reduced row-echelon matrix, using an auxiliary function that completes the conversion under the assumption (row-echelon-p-aux a m k), where $0 \leq k \leq m$:

```
(defun row-reduce-aux (a m k)
  (let ((i (row-with-nonzero-at-least-index a m k)))
    (if (and (natp k) (natp m) (< k m) i)
        (row-reduce-aux (row-reduce-step a m k i (first-nonzero (nth i a)))
                        m (1+ k))
      a)))

(defund row-reduce (a) (row-reduce-aux a (len a) 0))
```

The following confirms that this procedure produces the desired result:

```
(defthmd row-echelon-p-row-reduce
  (implies (and (natp m) (natp n) (fmatp a m n))
           (row-echelon-p (row-reduce a))))
```

We also note that row reduction does not alter a reduced row-echelon matrix:

```
(defthmd row-reduce-row-echelon-p
  (implies (and (posp m) (posp n) (fmatp a m n) (row-echelon-p a))
           (equal (row-reduce a) a)))
```

As an example, consider the following 4×5 matrix (a0):

```
DM !>(defun a0 () '((0 -3 -6 4 9) (-1 -2 -1 3 1) (-2 -3 0 3 -1) (1 4 5 -9 -7)))
```

In the first step in the row reduction of (a0), row 1 is divided by its leading nonzero entry, -1, and interchanged with row 0. The other entries in column 0 are then cleared:

```
DM !>(row-reduce-step (a0) 4 0 1 0)
((1  2  1 -3 -1)
 (0 -3 -6  4  9)
 (0  1  2 -3 -3)
 (0  2 4  -6 -6))
```

Row reduction of `(a0)` requires three executions of `row-reduce-step`:

```
DM !>(row-reduce (a0))
((1  0 -3  0  5)
 (0  1  2  0 -3)
 (0  0  0  1  0)
 (0  0  0  0  0))
```

We define the *row rank* of `a` to be the number of nonzero rows of `(row-reduce a)`:

```
(defun num-nonzero-rows (a)
  (if (consp a)
      (if (flist0p (car a)) 0 (1+ (num-nonzero-rows (cdr a))))
    0))

(defun row-rank (a) (num-nonzero-rows (row-reduce a)))
```

Note that `(row-reduce (a0))` has 3 nonzero rows:

```
DM !>(row-rank (a0))
3
```

Obviously, the row rank of an `m`×`n` matrix cannot exceed `m`:

```
(defthmd row-rank<=m
  (implies (and (fmatp a m n) (posp m) (posp n))
           (<= (row-rank a) m)))
```

Nor can the row rank exceed `n`. To see this, consider the list of indices of the leading 1s of the nonzero rows of a reduced row-echelon matrix `a`:

```
(defun lead-inds (a)
  (if (and (consp a) (not (flist0p (car a))))
      (cons (first-nonzero (car a)) (lead-inds (cdr a)))
    ()))

DM !>(lead-inds (row-reduce (a0)))
(0 1 3)
```

Clearly, the length of `(lead-inds a)` is the number of nonzero rows of `a`. Furthermore, `(lead-inds a)` is a strictly increasing sublist of `(ninit t)`. It follows that `(len (lead-inds a))` $\leq$ `n`. Consequently, the row rank of `a` is bounded by `n`:

```
(defthmd row-rank<=n
  (implies (and (fmatp a m n) (posp m) (posp n))
           (<= (row-rank a) n)))
```

We also note that if `(row-rank a)` = `n`, then `(lead-inds a)` is an increasing sublist of `(ninit n)` of length `n`, which implies that the two lists are equal:

```
(defthmd lead-inds-ninit
  (implies (and (fmatp a m n) (posp m) (posp n)
                (row-echelon-p a) (= (row-rank a) n))
           (equal (lead-inds a) (ninit n))))
```

Along with row reduction, there is an obvious analogous notion of *column reduction* and a corresponding definiton of the *column rank* of a matrix, which may alternatively be defined as the row rank of its transpose. As we shall show in Part III in the context of vector spaces, the row and column ranks of a matrix are always equal.

2.3 Row Reduction as Matrix Multiplication

Once we have identified the sequence of operations required to derive the reduced row-echelon form of an m × n matrix a, an alternative derivation may be achieved by applying the same operations to the m × m identity matrix and right-multiplying the result by a. To this end, a row operation is encoded as a list of length 3 or 4; the first member indicates the operation type (1, 2, or 3 as listed in the preceding subsection), and the others are the parameters of the operation. The following predicate characterizes an encoding of a row operation on a matrix of m rows:

```
(defund row-op-p (op m)
  (and (true-listp op)
       (case (car op)
         (1 (and (= (len op) 3) (fp (cadr op)) (not (= (cadr op) (f0)))
                 (natp (caddr op)) (< (caddr op) m)))
         (2 (and (= (len op) 4) (fp (cadr op))
                 (natp (caddr op)) (< (caddr op) m)
                 (natp (cadddr op)) (< (cadddr op) m)
                 (not (= (caddr op) (cadddr op)))))
         (3 (and (= (len op) 3) (natp (cadr op)) (< (cadr op) m)
                 (natp (caddr op)) (< (caddr op) m))))))
```

The function apply-row-op applies an encoded row operation to a matrix:

```
(defund apply-row-op (op a)
  (case (car op)
    ;(apply-row-op (list 1 c k) a) = (ero1 a c k)
    (1 (ero1 a (cadr op) (caddr op)))
    ;(apply-row-op (list 2 c j k) a) = (ero2 a c j k)
    (2 (ero2 a (cadr op) (caddr op) (cadddr op)))
    ;(apply-row-op (list 3 j k) a) = (ero3 a j k)
    (3 (ero3 a (cadr op) (caddr op)))))
```

A list of row operations is identified in the obvious way:

```
(defun row-ops-p (ops m)
  (if (consp ops)
      (and (row-op-p (car ops) m)
           (row-ops-p (cdr ops) m))
    (null ops)))
```

The function apply-row-ops applies a list of operations in sequence from left to right:

```
(defun apply-row-ops (ops a)
    (if (consp ops)
        (apply-row-ops (cdr ops) (apply-row-op (car ops) a))
      a))
```

By examining the definitions of row-reduce and its auxiliary functions, we construct the list of encodings of the operations that reduce a matrix a to reduced row-echelon form. The next four functions encode the lists of operations performed by clear-column, row-reduce-step, row-reduce-aux, and row-reduce, respectively:

```
(defun clear-column-ops (a k j m)
  (if (zp m) ()
    (if (= k (1- m))
        (clear-column-ops a k j (1- m))
      (cons (list 2 (f- (nth j (nth (1- m) a))) k (1- m))
            (clear-column-ops (ero2 a (f- (nth j (nth (1- m) a))) k (1- m))
                              k j (1- m))))))

(defund row-reduce-step-ops (a m k i j)
  (cons (list 1 (f/ (nth j (nth i a))) i)
        (cons (list 3 i k)
              (clear-column-ops (ero3 (ero1 a (f/ (nth j (nth i a))) i) i k)
                                k j m))))

(defun row-reduce-aux-ops (a m k)
  (let* ((i (row-with-nonzero-at-least-index a m k))
         (j (and i (first-nonzero (nth i a)))))
    (if (and (natp k) (natp m) (< k m) i)
        (append (row-reduce-step-ops a m k i j)
                (row-reduce-aux-ops (row-reduce-step a m k i j) m (1+ k)))

(defund row-reduce-ops (a) (row-reduce-aux-ops a (len a) 0))
```

The correctness of this encoding procedure is confirmed by the following:

```
(defthmd apply-row-reduce-ops
  (implies (and (fmatp a m n) (posp m) (posp n))
           (equal (apply-row-ops (row-reduce-ops a) a)
                  (row-reduce a))))
```

Returning to the example of Subsection 2.2, we find that the first step in the row reduction of `(a0)` involves five elementary operations:

```
DM !>(row-reduce-step-ops (a0) 4 0 1 0)
((1 -1 1) (3 1 0) (2 -1 0 3) (2 2 0 2) (2 0 0 1))

DM !>(apply-row-ops '((1 -1 1) (3 1 0) (2 -1 0 3) (2 2 0 2) (2 0 0 1)) (a0))
((1  2  1 -3 -1)
 (0 -3 -6  4  9)
 (0  1  2 -3 -3)
 (0  2  4 -6 -6))
```

The reader may wish to compute `(row-reduce-ops (a0))`, a list of length 15, and check that the lemma `apply-row-reduce-ops` holds in this case.

The `m`×`m` *elementary matrix* corresponding to a row operation is defined to be the result of applying the operation to the `m`×`m` identity matrix:

```
(defund elem-mat (op m) (apply-row-op op (id-fmat m)))
```

Application of a row operation is equivalent to left multiplication by the corresponding elementary matrix:

```
(defthmd elem-mat-row-op
  (implies (and (fmatp a m n) (row-op-p op m) (posp m) (posp n))
           (equal (fmat* (elem-mat op m) a) (apply-row-op op a))))
```

The product of the list of elementary matrices associated with the row reduction of a matrix is computed recursively by the function `row-reduce-mat`:

```
(defund row-ops-mat (ops m)
  (if (consp ops)
      (fmat* (row-ops-mat (cdr ops) m) (elem-mat (car ops) m))
    (id-fmat m)))

(defund row-reduce-mat (a) (row-ops-mat (row-reduce-ops a) (len a)))
```

It follows from `elem-mat-row-op` by induction that applying a sequence `ops` of row operations to `a` is equivalent to multiplication of `a` by `(row-ops-mat ops m)`:

```
(defthmd fmat*-row-ops-mat
  (implies (and (fmatp a m n) (posp m) (posp n)
                (row-ops-p ops m))
           (equal (fmat* (row-ops-mat ops m) a)
                  (apply-row-ops ops a))))
```

In particular, by `apply-row-reduce-ops`, row reduction of `a` is equivalent to multiplication by `(row-reduce-mat a)`:

```
(defthmd row-ops-mat-row-reduce
  (implies (and (fmatp a m n) (posp m) (posp n))
           (equal (fmat* (row-reduce-mat a) a) (row-reduce a))))
```

In our example, the product of the 15 elementary matrices corresponding to `(row-reduce-ops (a0))` is

```
DM !>(row-reduce-mat (a0))
(( 3/5 -3/5 -1/5 0)
 (-3/5  8/5 -4/5 0)
 (-1/5  6/5 -3/5 0)
 (  0    5   -2  1))
```

The conclusion of the lemma `row-ops-mat-row-reduce` may be readily verified for this case.

2.4 Invertibility

In this subsection, we focus on square matrices. Given an `n`×`n` matrix `a`, we seek an *inverse* of `a`, i.e., an `n`×`n` matrix `b` such that

```
(fmat* a b) = (fmat* b a) = (id-fmat n).
```

If such a matrix exists, then it is unique in the strong sense that it is the only left or right inverse of `a`. For example, if `(fmat* c a) = (id-fmat n)`, then

```
c = (fmat* c (id-fmat n))
  = (fmat* c (fmat* a b))
  = (fmat* (fmat* c a) b))
  = (fmat* (id-fmat n) b))
  = b,
```

and the same conclusion similarly follows from the assumption `(fmat* a c) = (id-fmat n)`. Thus, we have

```
(defthm inverse-unique
  (implies (and (fmatp a n n) (fmatp b n n) (fmatp c n n) (posp n)
                (= (fmat* a b) (id-fmat n)) (= (fmat* b a) (id-fmat n))
                (or (= (fmat* a c) (id-fmat n)) (= (fmat* c a) (id-fmat n))))
           (equal c b)))
```

Every elementary matix has an inverse:

```
(defund invert-row-op (op)
  (case (car op)
    (1 (list 1 (f/ (cadr op)) (caddr op)))
    (2 (list 2 (f- (cadr op)) (caddr op) (cadddr op)))
    (3 op)))

(defthmd fmat*-elem-invert-row-op
  (implies (and (row-op-p op n) (posp n))
           (and (equal (fmat* (elem-mat (invert-row-op op) n) (elem-mat op n))
                       (id-fmat n))
                (equal (fmat* (elem-mat op n) (elem-mat (invert-row-op op) n))
                       (id-fmat n)))))
```

Consequently, every product of elementary matrices has an inverse:

```
(defun invert-row-ops (ops)
  (if (consp ops)
      (append (invert-row-ops (cdr ops)) (list (invert-row-op (car ops))))
    ()))

(defthmd invert-row-ops-mat
  (implies (and (row-ops-p ops n) (posp n))
           (and (equal (fmat* (row-ops-mat (invert-row-ops ops) n)
                              (row-ops-mat ops n))
                       (id-fmat n))
                (equal (fmat* (row-ops-mat ops n)
                              (row-ops-mat (invert-row-ops ops) n))
                       (id-fmat n)))))
```

We shall show that `a` has an inverse iff `(row-rank a) = n` and that in this case, the inverse of `a` is `(row-reduce-mat a)`. Thus, we define

```
(defund invertiblep (a n) (= (row-rank a) n))
```

and

```
(defund inverse-mat (a) (row-reduce-mat a))
```

First we note that as a consequence of `lead-inds-ninit`, if `(invertiblep a n)`, then `(row-reduce a) = (id-fmat n)`:

```
(defthm row-echelon-p-id-fmat
  (implies (and (fmatp a n n) (posp n) (row-echelon-p a) (= (num-nonzero-rows a) n))
           (equal a (id-fmat n)))
```

Now let

```
p = (inverse-mat a) = (row-reduce-mat a) = (row-ops-mat (row-reduce-ops a) n),

q = (row-ops-mat (invert-row-ops (row-reduce-ops a)) n),
```

and

```
r = (fmat* p a) = (row-reduce a).
```

By `invert-row-ops-mat`, `(fmat* p q) = (fmat* q p) = (id-fmat n)`. Suppose `(row-rank r) = n`. By `row-echelon-p-id-fmat`, `(fmat* p a) = r = (id-fmat n)`, and by `inverse-unique`, `a = q`. Thus, `(invertiblep a n)` is a sufficient condition for the existence of an inverse:

```
(defthmd invertiblep-sufficient
  (implies (and (fmatp a n n) (posp n) (invertiblep a n))
         (let ((p (inverse-mat a)))
             (and (fmatp p n n)
                  (equal (fmat* a p) (id-fmat n))
                  (equal (fmat* p a) (id-fmat n))))))
```

To prove the necessity of `(invertiblep a n)`, suppose `(fmatp b n n)` and `(fmat* a b) = (id-fmat n)`. Then

```
(fmat* r (fmat* b q)) = (fmat* (fmt* p a) (fmat* b q))
                      = (fmat* p (fmat* (fmat* a b) q))
                      = (fmat* p q)
                      = (id-fmat n).
```

If `(invertiblep a n) = NIL`, then the last row of `r` is zero, and the same must be true of `(id-fmat n)`, a contradiction.

```
(defthmd invertiblep-necessary
  (implies (and (fmatp a n n) (fmatp b n n) (posp n) (= (fmat* a b) (id-fmat n)))
           (invertiblep a n)))
```

We note several consequences of the preceding results. First, an invertible matrix is the inverse of its inverse:

```
(defthmd inverse-inverse-mat
  (implies (and (fmatp a n n) (posp n) (invertiblep a n))
           (and (invertiblep (inverse-mat a) n)
                (equal (inverse-mat (inverse-mat a)) a))))
```

Cancellation laws hold for invertible matrices, e.g.,

```
(defthmd invertiblep-cancel
  (implies (and (fmatp a m n) (fmatp b m n) (fmatp p m m) (posp m) (posp n)
                (invertiblep p m))
           (iff (equal (fmat* p a) (fmat* p b))
                (equal a b))))
```

A matrix product is invertible iff each factor is invertible:

```
(defthmd invertiblep-factor
  (implies (and (fmatp a n n) (fmatp b n n) (posp n) (invertiblep (fmat* a b) n))
           (and (invertiblep a n) (invertiblep b n))))

(defthmd inverse-fmat*
  (implies (and (fmatp a n n) (fmatp b n n) (posp n)
                (invertiblep a n) (invertiblep b n))
           (and (invertiblep (fmat* a b) n)
                (equal (inverse-mat (fmat* a b))
                       (fmat* (inverse-mat b) (inverse-mat a))))))
```

Finally, we shall show that a is invertible iff its determinant is nonzero. First note that if `a` has inverse `b` and `(fdet a)` = 0, then by `fdet-multiplicative`,

```
(fdet (id-fmat n) n) = (fdet (fmat* a b)) = (f* (fdet a) (fdet b)) = 0,
```

a contradiction. Thus,

```
(defthmd invertiblep-fdet-not-zero
  (implies (and (fmatp a n n) (posp n) (invertiblep a n))
           (not (equal (fdet a n) (f0)))))
```

On the other hand, assume `(fdet a n)` $\neq 0$. By `fmat*-adjoint-fmat`,

```
(fmat* a (adjoint-fmat a n)) = (fmat-scalar-mul (fdet a n) (id-fmat n)),
```

which implies

```
(fmat* a (fmat-scalar-mul (f/ (fdet a n)) (adjoint-fmat a n)))
   = (fmat-scalar-mul (f/ (fdet a n)) (fmat* a (adjoint-fmat a n)))
   = (fmat-scalar-mul (f/ (fdet a n)) (fmat-scalar-mul (fdet a n) (id-fmat n)))
   = (id-fmat n),
```

and by `invertiblep-necessary`, `a` is invertible. This also establishes an alternative method for computing the inverse:

```
(defthmd fdet-not-invertiblep-zero
  (implies (and (fmatp a n n) (natp n) (> n 1) (not (equal (fdet a n) (f0))))
           (and (invertiblep a n)
                (equal (inverse-mat a)
                       (fmat-scalar-mul (f/ (fdet a n)) (adjoint-fmat a n))))))
```

3 Simultaneous Systems of Linear equations

Let `a` be an `m`×`n` matrix with `(entry i j a)` = $a_{\mathtt{i},\mathtt{j}}$ for $0 \leq \mathtt{i} < \mathtt{m}$ and $0 \leq \mathtt{j} < \mathtt{n}$, and let `b` = $(b_0 \ldots b_{\mathtt{m}-1})$ be an flist of length `m`. We seek an flist `x` = $(x_0 \ldots x_{\mathtt{n}-1})$ of length `n` such that for $0 \leq \mathtt{i} < \mathtt{m}$,

$$a_{\mathtt{i},0}x_0 + \ldots + a_{\mathtt{i},\mathtt{n}-1}x_{\mathtt{n}-1} = b_{\mathtt{i}}.$$

We shall refer to `a` as the *coefficient matrix* of this system of `m` linear equations in `n` unknowns. To express the system as a matrix equation, we define the *column matrix* corresponding to a given flist:

```
(defund col-mat (x) (transpose-mat (list x)))
```

The above equations are naturally expressed by the matrix equation in the following definition:

```
(defund solutionp (x a b) (equal (fmat* a (col-mat x)) (col-mat b)))
```

Let bc = (col-mat b), xc = (col-mat x), p = (row-reduce-mat a), ar = (fmat* p a), and br = (fmat* p bc). Left-multiplying the above equation by p yields the equivalent equation

$$\texttt{(fmat* ar xc) = br.} \tag{1}$$

Thus, we have

```
(defthmd reduce-linear-equations
  (implies (and (fmatp a m n) (posp m) (posp n) (flistnp b m) (flistnp x n))
           (let* ((bc (col-mat b)) (xc (col-mat x))
                  (p (row-reduce-mat a)) (ar (fmat* p a)) (br (fmat* p bc)))
             (iff (solutionp x a b)
                  (equal (fmat* ar xc) br)))))
```

Our objective, therefore, is to compute an n×1 column matrix xc that solves Equation (1), in which ar is an m×n reduced row-echelon matrix and br is an m×1 column matrix.

Let q = (num-nonzero-rows ar) = (row-rank a). We shall show that the existence of a solution to this equation is determined by whether the last m − q entries of br are all 0. This is true iff the following search returns NIL:

```
(defun find-nonzero (br q m)
  (if (and (natp q) (natp m) (< q m))
      (if (= (entry (1- m) 0 br) (f0))
          (find-nonzero br q (1- m))
        (1- m))
    ()))
```

Thus, we define

```
(defun solvablep (a b)
  (null (find-nonzero (fmat* (row-reduce-mat a) (col-mat b))
                      (row-rank a)
                      (len a))))
```

Suppose first that (find-nonzero br q m) = k ≠ NIL, so that (solvablep a b) = NIL. Then (row k ar) = (flistn0 n) and (entry k 0 br) ≠ 0. It follows that (entry k 0 (fmat* ar xc)) ≠ (nth k 0 br), and hence (fmat* ar xc) ≠ br. Combining this with reduce-linear-equations, we conclude that the system of equations has no solution:

```
(defthmd linear-equations-unsolvable-case
  (implies (and (fmatp a m n) (posp m) (posp n) (flistnp b m) (flistnp x n)
                (not (solvablep a b)))
           (not (solutionp x a b))))
```

Thus, we may assume (solvablep a b) = T. As a first step toward the solution, consider the matrices aq and bq consisting of the first q rows of ar and br, respectively, computed by the following:

```
(defun first-rows (q a)
  (if (zp q) ()
    (cons (car a) (first-rows (1- q) (cdr a)))))
```

It is easily shown that aq is a reduced row-echelon q×n matrix of row rank q and that (fmat* ar xc) = br iff (fmat* aq xc) = bq. Our objective, therefore, is to solve the equation (fmat* aq xc) = bq.

3.1 Uniquely Solvable Case

By row-rank<=n, q $\leq$ n. We first consider the case q = n. By row-echelon-p-id-fmat, aq = (id-fmat n) and (fmat* aq xc) = bq iff xc = bq. Combining this observation with first-rows-linear-equations and reduce-linear-equations, we conclude that there exists a unique solution in this case:;

```
(defthmd linear-equations-unique-solution-case
  (let* ((br (fmat* (row-reduce-mat a) (col-mat b)))
         (bq (first-rows n br)))
    (implies (and (fmatp a m n) (posp m) (posp n) (flistnp b m) (flistnp x n)
                  (solvablep a b) (= (row-rank a) n))
             (iff (solutionp x a b)
                  (equal x (col 0 bq))))))
```

Our results on cofactor expansion lead to an alternative method of solving a system of n linear equations in n unknowns in the case of a unique solution, known as Cramer's rule. Suppose m = n = q, so that a is an invertible n$\times$n matrix. Our objective is to compute, as a function of a and b, for each i $<$ n, the ith component (nth i x) of the unique x such that

$$\texttt{(fmat* a xc) = bc.} \tag{2}$$

We refer to the analogs of the results of [6, Sec. 5] that appear in fdet.lisp. In particular, we shall substitute a' = (replace-row (transpose-mat a) i b) for a in fdot-cofactor-fmat-row-fdet. Clearly, (row i a') = b. By cofactor-fmat-transpose,

```
(cofactor-fmat-row i a' n) = (cofactor-fmat-row i (transpose-mat a) n)
                           = (row i (cofactor-fmat (transpose-mat a) n))
                           = (row i (adjoint-fmat a n)),
```

and by fdet-transpose,

```
(fdet a' n) = (fdet (transpose-fmat (replace-col a i b)) n)
            = (fdet (replace-col a i b) n).
```

Thus, the substitution yields the following:

```
(fdot b (row i (adjoint-fmat a n))) = (fdet (replace-col a i b) n)).
```

Multiplying Equation (2) by (adjoint-fmat a n) yields

```
(fmat* (adjoint-fmat a n) (fmat* a xc)) = (fmat* (adjoint-fmat a n) bc).
```

But

```
(fmat* (adjoint-fmat a n) (fmat* a xc))
  = (fmat* (fmat* (adjoint-fmat a n) a) xc)
  = (fmat* (flist-scalar-mul (fdet a n) (id-fmat n)) xc)
  = (flist-scalar-mul (fdet a n) (fmat* (id-fmat n) xc))
  = (flist-scalar-mul (fdet a n) xc),
```

and hence

```
(flist-scalar-mul (fdet a n) xc) = (fmat* (adjoint-fmat a n) bc).
```

Equating the entries of these matrices in row i and column 0, we have

```
(f* (fdet a n) (nth i x)) = (fdot b (row i (adjoint-fmat a n)))
                          = (fdet (replace-col a i b) n),
```

which yields Cramer's rule:

```
(defthmd cramer
  (implies (and (fmatp a n n) (natp n) (> n 1) (invertiblep a n)
                (flistnp b n) (flistnp x n) (solutionp x a b)
                (natp i) (< i n))
           (equal (nth i x)
                  (f* (f/ (fdet a n))
                      (fdet (replace-col a i b) n)))))
```

3.2 General Solvable Case

In the remainder of this section, we treat the general case `(solvablep a b) = T` with arbitrary `q = (row-rank a)` $\leq$ `n`. The desired equation `(fmat* aq xc) = bq` holds iff for $0 \leq$ `i` $<$ `q`,

```
(nth i (fmat* aq xc)) = (nth i bq)
```

or equivalently,

```
(fdot (row i aq) x) = (car (nth i bq)).                    (3)
```

We shall split the dot product `(fdot (nth i aq) x)` into two sums, corresponding to the list `(lead-inds aq)` of leading indices and the list of remaining indices, which we call the *free indices*:

```
(defund free-inds (a n) (set-difference-equal (ninit n) (lead-inds a)))
```

In general, given a sublist `inds` of `(ninit n)` and two flists `r` and `x` of length `n`, the following function extracts and sums the terms of the dot product of `r` and `x` that correspond to the indices `inds`:

```
(defun fdot-select (inds r x)
  (if (consp inds)
      (f+ (f* (nth (car inds) r) (nth (car inds) x))
          (fdot-select (cdr inds) r x))
    (f0)))
```

In particular, `(fdot (row i aq) x)` may be expressed as the following sum:

```
  (f+ (fdot-select (lead-inds aq) (row i aq) x)
      (fdot-select (free-inds aq n) (row i aq) x))))).
```

Now since `(row i aq)` has a 1 at index `(nth i (lead-inds aq))` and a 0 at all other lead indices, the first of these two sums reduces to the single term `(nth (nth i (lead-inds aq)) x)`, and hence Equation (3) may be expressed as

```
(nth (nth i (lead-inds aq)) x)
  = (f+ (car (nth i bq))
        (f- (fdot-select (free-inds aq n) (row i aq) x))).
```

Thus, `x` is a solution of our system of equations iff this condition holds for all `i` $<$ `q`. This is checked recursively by the following function:

```
(defun solution-test-aux (x aq bq lead-inds free-inds k)
  (if (zp k) t
    (and (equal (nth (nth (1- k) lead-inds) x)
                (f+ (car (nth (1- k) bq))
                    (f- (fdot-select free-inds (nth (1- k) aq) x))))
         (solution-test-aux x aq bq lead-inds free-inds (1- k)))))

(defund solution-test (x a b n)
  (let* ((ar (row-reduce a))
         (br (fmat* (row-reduce-mat a) (col-mat b)))
         (q (num-nonzero-rows ar))
         (aq (first-rows q ar))
         (bq (first-rows q br))
         (lead-inds (lead-inds aq))
         (free-inds (free-inds aq n)))
    (solution-test-aux x aq bq lead-inds free-inds q)))
```

This provides a test to be applied to a candidate solution:

```
(defthmd linear-equations-solvable-case
  (implies (and (fmatp a m n) (posp m) (posp n) (flistnp b m) (flistnp x n)
                (solvablep a b))
           (iff (solutionp x a b)
                (solution-test x a b n))))
```

If `q` = `(len (lead-inds aq))` = `n`, then `(free-inds aq n)` = `NIL`, the equation

```
(nth (nth i l) x) = (f+ (car (nth i bq)) (f- (fdot-select f (nth i aq) x)))
```

reduces to

```
(nth i x) = (car (nth i bq),
```

`(solution-test-aux x aq bq l f q)` reduces to `x = (col 0 bq)`, and the last theorem reduces to the earlier result `linear-equations-unique-solution-case`.

Otherwise, `(free-inds aq n)` $\neq$ `NIL` and the components of `x` corresponding to the indices in `(lead-inds aq)` are determined by the components corresponding to `(free-inds aq n)`, which are unconstrained. Thus, there is a single solution corresponding to every assignment of values to the latter set of components, and hence infinitely many solutions. We shall revisit this result in Part III, where we show that in the homogeneous case, `b = (flistn0 n)`, the solutions form a vector space of dimension `n` $-$ `q`. A basis for this solution space will be provided by a formula derived from the function `solution-test`. .

4 Future Work

This formalization of linear algebra is a work in progress. In Part I, we developed the algebra of matrices over a commutative ring with unity and the theory of determinants. In this sequel, we have restricted our attention to matrices over a field in order to address the process of row reduction and its applications. To allow our results to be applied to an arbitrary ring or field, we have characterized each by an encapsulated set of constrained functions.

There is progress to report on a planned Part III, which begins with another encapsulation that formalizes the notion of an abstract finite-dimensional vector space over the field `F`. The constrained functions of this encapsulation naturally include a predicate that recognizes vectors in the space, the operations of vector addition and scalar multiplication, and the constant 0 vector. Two additional functions embody the requirement of finite dimensionality: (1) a constant list of vectors of unspecified length that serves as a canonical basis, and (2) a function that returns the coordinates of a given vector with respect to this basis. Thus, whenever we define a concrete vector space, we are obligated to identify a basis for it. This establishes a tight connection between vector spaces and matrices: a list of vectors may be identified by the matrix of coordinates of its members. As an unexpected application of row reduction, this connection provides an algorithmic definition of the basic notion of linear independence without use of quantifiers: a list of vectors is linearly independent iff the row rank of its coordinate matrix is the length of the list.

The reader may have noticed that the definition of the basic notion of row equivalence is omitted from our treatment of row reduction. Recall that two matrices are said to be row equivalent if one may be derived from the other by a sequence of elementary row operations. This definition could be formalized in ACL2 using the support for existential quantification provided by `defun-sk`, and the properties of an equivalence relation could be derived from the results of Section 2. We could also prove the important theorem that distinct reduced row-echelon matrices cannot be row equivalent. However, since its most expedient proof is based on vector spaces (in particular, the row space of a matrix), this result is postponed to Part III. Note that it provides an alternative definition of row equivalence that avoids quantification: two matrices `a` and `b` are row equivalent iff `(row-reduce a) = (row-reduce b)`. Since we prefer this algorithmic formulation, the entire topic is deferred to Part III.

Other topics to be addressed in the sequel include linear transformations and diagonalization. As discussed in Part I, a factor in our decision to develop the algebra of matrices over an arbitrary commutative ring rather than a field is that this allows us to define the characteristic polynomial of a square matrix over the field `F` as the determinant of a certain matrix over the polynomial ring $\mathtt{F}[t]$. A related objective is the proof of the Cayley-Hamilton Theorem (every square matrix over a commutative ring is a root of its own characteristic polynomial), which has wide-ranging applications in other areas of mathematics.

This project is part of a broader effort in the formalization of algebra, which began with group theory [3, 4, 5] and will continue beyond linear algebra. Our next targeted area of investigation will be Galois theory, which we hope eventually to apply to the study of algebraic number fields. These intended applications guided our choices of formalization schemes for the basic algebraic structures. Since we are interested in infinite rings and fields, the encapsulation approach seems to be the only viable representation scheme for these structures provided by the ACL2 logic. The disadvantages of not being able to refer to such a structure as an ACL2 object are obvious. On the other hand, since the groups of primary interest are finite, our investigation of group theory is limited to the finite case. Under this restriction, a group is conveniently represented as an object characterized by a predicate defined as an ACL2 function. There are, however, infinite groups of interest, which are not accommodated by our theory. For example, the general linear group of invertible matrices over a field, which is in general an infinite structure, would otherwise have provided an interesting example and another connection between group theory and linear algebra.

Although we have no immediate plans to apply this theory beyond the realm of pure mathematics, its potential utility in the formal verification of hardware and software applications is limitless. Linear algebra is central to the rapidly advancing fields of machine learning and neural networks [1], providing essential tools for data reprersentation and manipulation. Along with finite group theory, it is also important to a variety of cryptographic algorithms [2]. It is our hope that ACL2 users who are interested in pursuing such applications may find our results useful.

References

[1] Sahar Halim (2020): *Application of Linear Algebra in Machine Learning*. International Research Journal of Engineering and Technology 7(2).

[2] Yuling Qian (2023): *Application of Modern Algebra in Cryptography*. Theoretical and Natural Science 10(1), doi:10.54254/2753-8818/10/20230304.

[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] 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, doi:10.4204/EPTCS.393.4.

[5] David M. Russinoff (2023): *A Formalization of Finite Froup Theory: Part III*. In: *ACL2 2023: 18th International Workshop on the ACL2 Theorem Prover and its Applications*, Austin, Texas, doi:10.4204/EPTCS.393.5.

[6] David M. Russinoff (2025): *A Formalization of Elementary Linear Algebra: Part I*. In: *ACL2 2025: 19th International Workshop on the ACL2 Theorem Prover and its Applications*, Austin, Texas.

A Proof of the Schröder-Bernstein Theorem in ACL2

Grant Jurgensen
Kestrel Institute
Palo Alto, California
grant@kestrel.edu

The Schröder-Bernstein theorem states that, for any two sets P and Q, if there exists an injection from P to Q and an injection from Q to P, then there must exist a bijection between the two sets. Classically, it follows that the ordering of the cardinal numbers is antisymmetric. We describe a formulation and verification of the Schröder-Bernstein theorem in ACL2 following a well-known proof, introducing a theory of *chains* to define a non-computable witness.

1 Introduction

In this paper we present a formulation and verification of the Schröder-Bernstein theorem in ACL2. To our knowledge, this is the first proof of the theorem in the Boyer-Moore family of theorem provers, although it has been verified in a number of other theorem provers, including Isabelle [8], Rocq (formerly Coq) [4], Lean [1], Metamath [7], and Mizar [9].

This paper is organized as follows. In Section 2, we outline the mathematical background and the general proof which will serve as the basis for the ACL2 formalization. In Section 3.1, we describe the formulation of the theorem's premises in ACL2. In Section 3.2, we describe our approach to defining function inverses and present a macro to quickly introduce inverses and their essential theorems. In Section 3.3, we present a theory of *chains*, mirroring the concept to be defined in the informal proof sketch. Finally, Section 3.4 defines the non-computable bijective function and summarizes the intermediate lemmas and final theorems which conclude the proof of the Schröder-Bernstein theorem.

The full proof and surrounding theory can be found in the ACL2 community books under `projects/schroeder-bernstein`.

2 The Informal Proof

Given two injective functions $f : P \to Q$ and $g : Q \to P$, the Schröder-Bernstein theorem states there must exist a bijection $h : P \to Q$. Before presenting the formalization within ACL2, we begin with a proof sketch based upon [3], which in turn closely follows Julius König's original proof [6].

2.1 A Theory of Chains

This proof proceeds from a theory of *chains*. For convenience, let us assume sets P and Q are disjoint [1] . We define a chain $C \subseteq P \cup Q$ as a set of elements which are mutually reachable via repeated application of f and g, or their inverses. So the element $p \in P$ is a member of the following chain.

$$\{\ldots,\ f^{-1}(g^{-1}(p)),\ g^{-1}(p),\ p,\ f(p),\ g(f(p)),\ \ldots\}$$

[1] To generalize the argument to arbitrary sets, we need only tag elements reflecting their association with one of the two sets. Indeed, we employ this strategy in the ACL2 formalization.

Gamboa and Manolios (Eds):
ACL2 Workshop 2025
EPTCS 423, 2025, pp. 36–45, doi:10.4204/EPTCS.423.3

Similarly, $q \in Q$ belongs to the chain:

$$\{\ldots, g^{-1}(f^{-1}(q)), f^{-1}(q), q, g(q), f(g(q)), \ldots\}$$

Every chain falls in one of a number of categories:

1. **Cyclic chains**: After some finite number of steps, the chain cycles back to a previous element.
2. **Infinite chains**: All acyclic chains are (countably) infinite. Infinite chains all extend infinitely in the "rightward" direction and may be further subdivided into two categories:
   (a) **Non-stoppers**: Such chains extend infinitely in the leftward direction in addition to the rightward direction.
   (b) **Stoppers**: Such chains do *not* extend infinitely leftward and may therefore be said to possess an *initial* element. On such an element, neither f^{-1} nor g^{-1} is defined (i.e., the element is not in the image of f or g).

An ordering on chain elements is implied above which follows the order in which the elements of the two example chains were enumerated. This simple ordering may be more rigorously defined as the reflexive-transitive closure of the relation defined by the following two inference rules.

$$\frac{p \in P}{p \sqsubseteq f(p)} \qquad \frac{q \in Q}{q \sqsubseteq g(q)}$$

This order is neither symmetric nor antisymmetric in general and is therefore a preorder. (On infinite chains, however, the order is antisymmetric and therefore a partial order. On cyclic chains, it is symmetric and therefore an equivalence relation.) Let $chain(x)$ denote the chain to which x belongs. We note that, for arbitrary $x, y \in P \cup Q$, the equality $chain(x) = chain(y)$ holds if and only if $x \sqsubseteq y$ or $y \sqsubseteq x$. It follows that the set of chains partition $P \cup Q$.

Note that an initial element is minimal with respect to this ordering. That is, value i is initial if and only if $x \sqsubseteq i$ implies $x = i$ for arbitrary x. This definition is equivalent to the one given above.

An initial element may reside either in P or Q. We further subdivide the category of stopper chains, referring to chains with initial elements in P as "P-stoppers" and those with initial elements in Q as "Q-stoppers".

Lemma 1. *The initial element of a chain is unique.*

Proof. This fact follows immediately from the minimality of initial elements. Let x and y be initial within the same chain. As noted above, we have $x \sqsubseteq y$ or $y \sqsubseteq x$ since the two share a chain. Without loss of generality, assume $x \sqsubseteq y$. Then by the minimality of initial element y, we have $x = y$. □

2.2 Definition and Proof of the Bijection

With the above theory of chains established, we are able to define our bijection. Let $stoppers_Q$ denote the set of Q-stoppers. Then we define our proposed bijection h:

$$h(p) = \begin{cases} g^{-1}(p) & \text{if } chain(p) \in stoppers_Q \\ f(p) & \text{otherwise} \end{cases}$$

The decision to use this particular definition of h is, in part, arbitrary. When $chain(p)$ is cyclic or a non-stopper, either f or g^{-1} are possible definitions. We choose to bias toward the use of f, which will be more convenient in the subsequent ACL2 formalization.

We begin with a few prerequisite lemmas before proceeding to establish bijectivity.

Lemma 2. *Let $p \in P$ and $chain(p) \in stoppers_Q$. Then p is in the image of g.*

Proof. By the definition of a Q-stopper, the initial element of $chain(p)$ resides in Q. Since the initial element is unique (Lemma 1) and $p \notin Q$, p must not be initial. Therefore, it is by definition in the image of g. □

Lemma 3. *Let $q \in Q$ and $chain(q) \notin stoppers_Q$. Then q is in the image of f.*

Proof. If $chain(q)$ has an initial element, then the initial element must be in P. Since $q \notin P$, it is not initial. If $chain(q)$ does not have an initial element, then clearly q is again not initial. By definition then, q is in the image of f. □

These lemmas establish when we may safely take the inverse of f and g. Lemma 2 in particular shows that the first case of our bijection h is well-defined.

Lemma 4. *Let $p \in P$. Then $chain(h(p)) = chain(p)$.*

Proof. Either $h(p) = g^{-1}(p)$ or $h(p) = f(p)$. By definition, p is in the same chain as $f(p)$ as well as $g^{-1}(p)$, if it is defined. □

Lemma 5 (Injectivity of h)**.** *Let $p_0, p_1 \in P$, where $h(p_o) = h(p_1)$. Then $p_0 = p_1$.*

Proof.
Case 1: $h(p_0)$ is in a Q-stopper.

By equality, $h(p_1)$ is also in a Q-stopper. By Lemma 4, so are p_0 and p_1. By definition, we have $h(p_0) = g^{-1}(p_0)$ and $h(p_1) = g^{-1}(p_1)$. From $h(p_0) = h(p_1)$, we get $g^{-1}(p_0) = g^{-1}(p_1)$. Applying g yields $p_0 = p_1$.

Case 2: $h(p_0)$ is not in a Q-stopper.

$h(p_1)$, p_0, and p_1 are also not in Q-stoppers. By definition, we then have $h(p_0) = f(p_0)$ and $h(p_1) = f(p_1)$. From $h(p_0) = h(p_1)$, we get $f(p_0) = f(p_1)$. By injectivity of f, we have $p_0 = p_1$. □

Lemma 6 (Surjectivity of h)**.** *Let $q \in Q$. Then there exists $p \in P$ such that $h(p) = q$.*

Proof.
Case 1: q is in a Q-stopper.

Then $g(q)$ is also in a Q-stopper by definition. Let $p = g(q)$. Then:

$$\begin{aligned} h(p) &= h(g(q)) \\ &= g^{-1}(g(q)) \\ &= q \end{aligned}$$

Case 2: q is not in a Q-stopper.

By Lemma 3, $f^{-1}(q)$ is well-defined. Since q is not in a Q-stopper, neither is $f^{-1}(q)$. Let $p = f^{-1}(q)$. Then:

$$\begin{aligned} h(p) &= h(f^{-1}(q)) \\ &= f(f^{-1}(q)) \\ &= q \end{aligned}$$

□

Theorem 1 (Schröder-Bernstein)**.** *h is bijective.*

Proof. By Lemma 5 and Lemma 6. □

3 ACL2 Formalization

3.1 Setup

To verify the Schröder-Bernstein theorem within ACL2, we closely follow the informal proof outlined in the previous section. We begin by introducing our "sets" as well as their injections. Since ACL2 is first-order [2] , we do not explicitly quantify over either. Instead, we introduce arbitrary predicates (representing the sets) and the injections between them via an `encapsulate` event [3] .

```
(encapsulate
  (((f *) => *)
   ((g *) => *)
   ((p *) => *)
   ((q *) => *))

  (local (define p (x) (declare (ignore x)) t))
  (local (define q (x) (declare (ignore x)) t))

  (local (define f (x) x))
  (local (define g (x) x))

  (defrule q-of-f-when-p
    (implies (p x)
             (q (f x))))

  (defrule injectivity-of-f
    (implies (and (p x)
                  (p y)
                  (equal (f x) (f y)))
             (equal x y))
    :rule-classes nil)

  (defrule p-of-g-when-q
    (implies (q x)
             (p (g x))))
```

[2] ACL2 offers limited second-order functionality through `apply$` [5]. However, `apply$` only operates on objects corresponding to a proper subset of ACL2's functions syntactically determined to be "tame." We might also have used SOFT [2] to simulate second-order functions.

[3] This ACL2 code snippet, as well as many of the following, are modified slightly for brevity. In particular, we elide proof hints, `xargs`, and returns specifications.

```
(defrule injectivity-of-g
  (implies (and (q x)
                (q y)
                (equal (g x) (g y)))
           (equal x y))
  :rule-classes nil))
```

Functions `p` and `q` correspond to the sets P and Q and are totally unconstrained. Although we interpret them as predicates, there is no need to constrain them to be strictly boolean-valued. Similarly, the ACL2 functions `f` and `g` correspond to the mathematical functions f and g in our informal proof. For these functions, we introduce two constraints each. First, since ACL2 functions are total, we require a theorem confirming the output of the function is in the codomain given that the input is in the intended domain (theorems `q-of-f-when-p` and `p-of-g-when-q`). Second, we establish the function's injectivity within said domain (theorems `injectivity-of-f` and `injectivity-of-g`). In general, subsequent theorems concerning `f` and `g` only characterize the functions applied to their respective domains.

3.2 Function Inverses

Before we can define our bijective witness, we must define a variety of auxiliaries, starting with our function inverses. Of course, the inverses of arbitrary functions are not computable. So, we must define our inverses via `defchoose` events. To quickly introduce such inverses and their essential theorems, we define a macro, `definverse`. As an example of what `definverse` produces, the declaration `(definverse f :domain p :codomain q)` emits the following definitions:

```
(define is-f-inverse (inv x)
  (and (p inv)
       (q x)
       (equal (f inv) x)))

(defchoose f-inverse (inv) (x)
  (is-f-inverse inv x))

(define in-f-imagep (x)
  (is-f-inverse (f-inverse x) x))
```

While f^{-1} is only defined on the image of f, the ACL2 function `f-inverse` is total. However, recall that a function introduced by `defchoose` will be unconstrained when the predicate on which it is defined is unsatisfiable. So the value of `(f-inverse x)` is unspecified when `x` is outside the image of `f`. Thus, we are only able to characterize `(f-inverse x)` when `(in-f-imagep x)` can be established.

In addition to the definitional events above, a number of theorems are also generated pertaining to the domain and codomain of the inverse function as well as the identity of the left and right compositions of the original function with its inverse. From the same example, we have:

```
(defrule in-f-imagep-of-f-when-p
  (implies (p x)
           (in-f-imagep (f x))))
```

```
(defrule p-of-f-inverse-when-in-f-imagep
  (implies (in-f-imagep x)
           (p (f-inverse x))))

;; Left inverse
(defrule f-inverse-of-f-when-p
  (implies (p x)
           (equal (f-inverse (f x))
                  x)))
;; Right inverse
(defrule f-of-f-inverse-when-in-f-imagep
  (implies (in-f-imagep x)
           (equal (f (f-inverse x))
                  x)))
```

We define the inverses of both `f` and `g` with this `definverse` macro.

3.3 The Theory of Chains

To define chains, we begin by defining chain elements, recognized by the `chain-elemp` predicate. A chain element is represented as a tagged value residing in either `p` or `q`, depending on the tag. This tagging is required to avoid the assumption of disjointedness present in the informal proof. We refer to a chain element's tag as its *polarity*. The ACL2 predicate `(polarity x)` holds when chain element `x` belongs to `p`. Otherwise, a valid chain element belongs to `q`.

```
(define chain-elemp (x)
  (and (consp x)
       (booleanp (car x))
       (if (car x)
           (and (p (cdr x)) t)
         (and (q (cdr x)) t))))

;; Construct a chain element
(define chain-elem (polarity val)
  (cons (and polarity t) val))

;; Get the polarity of a chain element
(define polarity ((elem consp))
  (and (car elem)
       t))

;; Get the value of a chain element
(define val ((elem consp))
  (cdr elem))
```

Since chains may be infinite, we cannot construct them explicitly by enumerating their elements. Instead, we define a non-computable equivalence, `chain=`, which relates chain elements belonging to

the same chain [4] .

```
(define chain= ((x consp) (y consp))
  (if (and (chain-elemp x)
           (chain-elemp y))
      (or (chain<= x y)
          (chain<= y x))
    (equal x y)))
```

When `x` and `y` are not chain elements, we fall back to regular equality to ensure that the function is an equivalence relation for all inputs. The `chain<=` function, which appears in our definition of `chain=`, corresponds to the ordering relation $\sqsubseteq$ discussed in Section 2. Formally, we define it using the following existential quantification.

```
(define-sk chain<= ((x consp) y)
  (exists n
    (equal (chain-steps x (nfix n))
           y)))
```

Here, `(chain-steps x n)` yields the chain element obtained from taking `n` steps "right" along the chain (applying either `f` or `g`, depending on the polarity), starting from the element `x`. We define it as follows.

```
(define chain-step ((elem consp))
  (let ((polarity (polarity elem)))
    (chain-elem (not polarity)
                (if polarity
                    (f (val elem))
                  (g (val elem))))))

(define chain-steps ((elem consp) (steps natp))
  (if (zp steps)
      elem
    (chain-steps (chain-step elem) (- steps 1))))
```

Beyond comparing whether two elements reside in the same chain, we must also characterize initial chain elements and Q-stoppers.

```
(define initialp ((elem consp))
  (if (polarity elem)
      (not (in-g-imagep (val elem)))
    (not (in-f-imagep (val elem)))))

(define initial-wrt ((initial consp) (elem consp))
  (and (chain-elemp initial)
       (initialp initial)
       (chain<= initial elem)))
```

[4] It would be straightforward to identify chains with some canonical element of the chain, chosen arbitrarily via a `defchoose` with the `:strengthen t` keyword argument. This step is, however, unnecessary for our proof of the Schröder-Bernstein theorem.

```
(defchoose get-initial (initial) (elem)
  (initial-wrt initial elem))

(define exists-initial ((elem consp))
  (initial-wrt (get-initial elem) elem))
```

In Section 2, we provided two equivalent definitions of initial elements. In the ACL2 formalization, we opt for the first definition, based on membership within the images of f and g (i.e., the existence of an inverse). The alternative definition, based on the minimality of initial elements, might have been employed via a Skolem function like so:

```
(define-sk initialp-alt ((elem consp))
  (forall x
    (implies (and (chain-elemp x)
                  (chain<= x elem))
             (equal elem x)))))
```

Such a definition is appealing in its conceptual simplicity. However, the introduction of yet another quantifier and Skolem function beyond those already required would further burden the proofs with necessary `:use` hints. Instead, we prefer to adopt the original definition and prove the minimality of initial elements as a consequence:

```
(defrule chain<=-of-arg1-and-initial
  (implies (and (chain-elem-p x)
                (initial-p initial))
           (equal (chain<= x initial)
                  (equal x initial))))
```

Similarly, `initial-wrt` (pronounced "initial with respect to") might have been defined in terms of `chain=`. But, as implied by the above, `(chain<= initial x)` and `(chain= initial x)` are equivalent when `initial` is initial. Therefore, we choose the stronger definition.

Finally, we may define membership of a chain element within a Q-stopper.

```
(define in-q-stopper ((elem consp))
  (and (exists-initial elem)
       (not (polarity (get-initial elem)))))
```

3.4 The Bijective Witness

Our bijective witness is now easily defined, following the piecewise definition h from the informal proof.

```
(define sb-witness (x)
  (if (in-q-stopper (chain-elem t x))
      (g-inverse x)
    (f x)))
```

We prove key theorems regarding when a chain element is necessarily in the image of `f` or `g`, mirroring Lemma 2 and Lemma 3 of the proof sketch.

```
(defrule in-g-imagep-when-in-q-stopper
  (implies (and (in-q-stopper elem)
                (polarity elem))
           (in-g-imagep (val elem))))
```

```
(defrule in-f-imagep-when-not-in-q-stopper
  (implies (and (chain-elemp elem)
                (not (in-q-stopper elem))
                (not (polarity elem)))
           (in-f-imagep (val elem))))
```

Similarly, we prove the analogue of Lemma 4, which shows `sb-witness` preserves chain membership.

```
(defrule chain=-of-sb-witness
  (implies (p x)
           (chain= (chain-elem t x)
                   (chain-elem nil (sb-witness x)))))
```

Finally, we prove the following three theorems which establish the bijectivity of `sb-witness` and therefore conclude our verification of the Schröder-Bernstein theorem.

```
(defrule q-of-sb-witness-when-p
  (implies (p x)
           (q (sb-witness x))))

(defrule injectivity-of-sb-witness
  (implies (and (p x)
                (p y)
                (equal (sb-witness x)
                       (sb-witness y)))
           (equal x y)))

(define-sk exists-sb-inverse (x)
  (exists inv
    (and (p inv)
         (equal (sb-witness inv) x))))

(defrule surjectivity-of-sb-witness
  (implies (q x)
           (exists-sb-inverse x)))
```

4 Conclusion

We have presented a formulation and verification of the Schröder-Bernstein theorem within ACL2. We started with an informal illustration of one of the theorem's well-known proofs. We then demonstrated how this proof mapped into the logic of ACL2. We introduced our generic "sets" via predicates, locally encapsulated with their two generic injections. We then defined function inverses as well as our theory of chains using Skolem functions. For the former, we introduced the `definverse` macro to quickly define function inverses. Finally, we presented the bijective witness, some key intermediate lemmas corresponding to steps in the informal proof, and then the three theorems which together establish bijectivity within the domain, thereby completing the proof of the Schröder-Bernstein theorem.

References

[1] Mario Carneiro: *Mathlib Documentation: Schröder-Bernstein theorem, well-ordering of cardinals*. https://leanprover-community.github.io/mathlib4_docs/Mathlib/SetTheory/Cardinal/SchroederBernstein.html. Accessed: 2025-01-21.

[2] Alessandro Coglio (2015): *Second-Order Functions and Theorems in ACL2*. International Workshop on the ACL2 Theorem Prover and Its Applications, pp. 17–33, doi:10.4204/EPTCS.192.3.

[3] Michael George: *Lecture Notes, CS 2800*. Available at https://www.cs.cornell.edu/courses/cs2800/2017fa/lectures/lec14-cantor.html. Accessed: 2025-01-07.

[4] Hugo Herbelin (1999): *GitHub Repository: rocq-archive/schroeder*. https://github.com/rocq-archive/schroeder. Accessed: 2025-01-21.

[5] Matt Kaufmann & J Strother Moore (2018): *Limited Second-Order Functionality in a First-Order Setting*. Journal of Automated Reasoning 64, pp. 391–422, doi:10.1007/s10817-018-09505-9.

[6] Julius König (1906): *Sur la Théorie des Ensembles*. Comptes Rendus Hebdomadaires des Séances de l'Académie des Sciences 143, pp. 110 – 112.

[7] Norman Megill & Jim Kingdon: *MetaMath Proof Explorer: Theorem sbth*. https://us.metamath.org/mpeuni/sbth.html. Accessed: 2025-01-21.

[8] Lawrence C. Paulson (1995): *Set Theory for Verification: II. Induction and Recursion*. Journal of Automated Reasoning 15, pp. 167–215, doi:10.1007/BF00881916.

[9] Piotr Rudnicki & Andrzej Trybulec (1997): *Fixpoints in Complete Lattices*. Formalized Mathematics 6(1), pp. 109–115. Available at http://fm.mizar.org/1997-6/pdf6-1/knaster.pdf.

RV32I in ACL2

Carl Kwan
The University of Texas at Austin
carlkwan@cs.utexas.edu

We present a simple ACL2 simulator for the RISC-V 32-bit base instruction set architecture, written in the operational semantics style. Like many other ISA models, our RISC-V state object is a single-threaded object and we prove read-over-write, write-over-write, writing-the-read, and state well-formedness theorems. Unlike some other models, we separate the instruction decoding functions from their semantic counterparts. Accordingly, we verify encoding / decoding functions for each RV32I instruction, the proofs for which are entirely automatic.

RISC-V is a popular open-source instruction set architecture (ISA) designed to be simple, flexible, and scalable. Unlike proprietary ISAs, RISC-V is free to use and modify, facilitating wide adoption across industries. A 2022 report suggests "there are more than 10 billion RISC-V cores in the market, and tens of thousands of engineers working on RISC-V initiatives globally" [12]. This motivates our development of a formal RISC-V simulator: to analyze and ensure the correctness of RISC-V hardware and software designs.

We present an executable ACL2 formal model of the 32-bit RISC-V base instruction set architecture (RV32I) [4], formalized by way of operational semantics [11, 10], and consisting of:

- a state object, formalized as an ACL2 single-threaded object (stobj) [5, 1];
- instruction semantic functions for all 37 RV32I (non-environment) instructions;
- step / run functions for simulating one or more fetch-decode-execute cycles;
- standard read-over-write, write-over-write, writing-the-read, and state well-formedness theorems;
- instruction encoding / decoding functions, and their inversion proofs;
- memory conversion theorems for execution using a byte-addressable model and proving theorems in word-addressable contexts.

Figure 1 summarizes the executable components in our model. Our stobj state object `rv32` consists of:

- 32 registers, one of which is hardwired to 0 and 31 general-purpose registers;
- 1 program counter register to hold the address of the current instruction;
- 2^{32} bytes of addressable memory;
- a model state parameter `ms` used for debugging (not an official part of the RISC-V specification).

We prove a standard collection of theorems involving the behaviour of `rv32` under its access and update functions. These are the read-over-write, write-over-write, writing-the-read, and state well-formedness theorems. A more comprehensive treatment of these theorems can be found in the description of the ACL2 x86 simulator [6, p. 37], so we discuss only one example of read-over-write in this document. Reading a byte of memory in `rv32` at address `i` is made by `(rm08 i rv32)`; updating `rv32` at the same memory address with value `v` is performed by `(wm08 i v rv32)`, which returns a new state object. The following read-over-write theorem roughly states that if we read a byte from a 32-bit memory address `i` after updating memory address `i` with a 8-bit value `v`, then we obtain `v` (regardless of what value was at address `i` previously):

Gamboa and Manolios (Eds):
ACL2 Workshop 2025
EPTCS 423, 2025, pp. 46–50, doi:10.4204/EPTCS.423.4

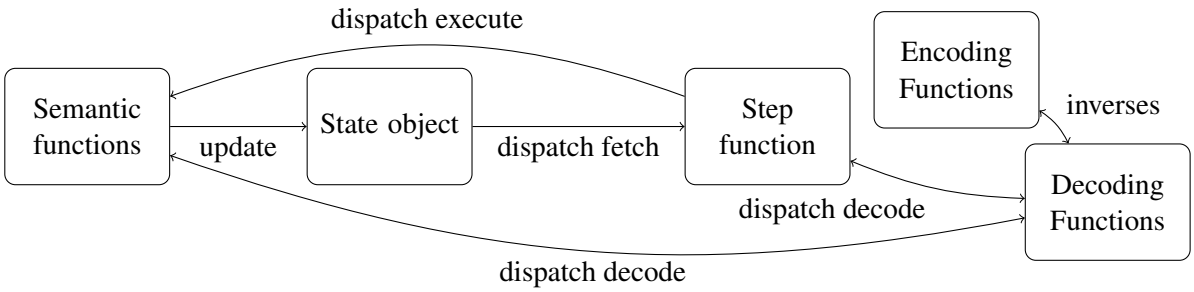

Figure 1: Overview of the ACL2 RV32I model.

```
(defthm rm08-wm08 (implies (and (n32p i) (n08p v)) (equal (rm08 i (wm08 i v rv32)) v)))
```

Similar theorems, for every standardized theorem sort, are proven for every parameter of `rv32`.

The RISC-V specification intends a byte-addressable memory model; however, because RV32I instructions (and many other 32-bit extensions) are standardized to 32-bits and required to be aligned on a four-byte boundary, it is sometimes easier to reason about memory as if it were word-addressable. Our base model uses a byte-addressable memory model, but we also formalize and verify functions for accessing memory as if it were word-addressable. For example, `rm32` is a function defined similarly to `rm08`, but directly obtains 4 bytes using state access functions. The following theorem states that reading a word from memory at address `addr` is equivalent to reading 4 successive bytes using `rm08` starting at `addr` and concatenating the result:

```
(defthmd rm32-from-successive-bytes
 (equal (rm32 addr rv32)                                        ;; read a word at addr
        (n32 (logior      (rm08      addr   rv32)              ;; read a byte at addr
                     (ash (rm08 (+ 1 addr)  rv32)  8)          ;; read a byte at addr + 1 and shift  8 bits
                     (ash (rm08 (+ 2 addr)  rv32) 16)          ;; read a byte at addr + 2 and shift 16 bits
                     (ash (rm08 (+ 3 addr)  rv32) 24)))))      ;; read a byte at addr + 3 and shift 24 bits
```

This enables us to treat the memory in our `rv32` state object as if it were word-addressable, but remain logically consistent to a byte-addressable model.

To simulate the execution of an RV32I instruction, we define instruction semantic functions, which directly update the `rv32` state object. These functions are called by a "step" function (see snippet below) that performs the "fetch" stage by obtaining the instruction to be executed from the memory of `rv32`:

```
(define rv32-step ((rv32 rv32p))    ;; Takes an rv32 machine state object
 (b* ((PC     (xpc     rv32))       ;; Fetch PC
      (instr  (rm32 PC rv32))       ;; Fetch instruction (32-bit value) from memory
      (opcode (get-opcode instr))   ;; Decode opcode from instruction
      (funct3 (get-funct3 instr))   ;; Decode funct3 from instruction
      (funct7 (get-funct7 instr)))  ;; Decode funct7 from instruction
     (case opcode                   ;; Pattern match on opcode
      (#b0110011                    ;; opcode for R-type instructions
       (case funct3                 ;; Pattern match on funct3
        (#x0                        ;; funct3 for integer ADD / SUB instructions
         (case funct7               ;; Pattern match on funct7
          (#x0  (rv32-add rv32))    ;; funct7 for ADD instruction, offload to rv32-add semantic function
       ...                          ;; Pattern matching and semantic function calls for other instructions
```

Note that the step function offloads the decoding of the instruction's `opcode`, `funct3`, and `funct7` (if

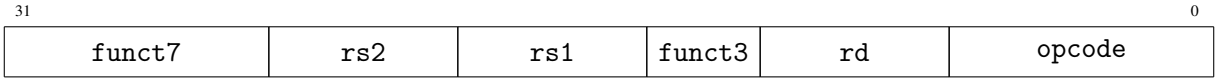

Figure 2: RV32I R-type instruction format.

applicable), which are fields that uniquely determine the instruction to be executed, to a layer of decoding functions (e.g. `get-opcode`, etc.). Actual execution is dispatched to the instruction semantic functions. Similarly to `rv32-step`, semantic functions also offload the decoding of any registers, memory addresses, or immediate values involved to more decoding functions. Finally, semantic functions update the `rv32` state accordingly.

There are 6 core instruction formats (R-type, I-type, S-type, B-type, U-type, and J-type). The format for one of them (R-type) is visualized by Figure 2. These formats dictate the role of the particular bits within an instruction. We formalize decoding functions for obtaining the appropriate bits as part of the decode stage, e.g. the function call `(get-opcode instr)` obtains bits 0–7 from the 32-bit value `instr`. Other types of instructions may have differing fields and field sizes, for which we also formalize decoding functions. Conversely, we also formalize encoding functions for each RV32I instruction, e.g. `(asm-add rs1 rs2 rd)` assembles the 32-bit instruction which stores the sum of the values from registers `rs1` and `rs2` into the destination register `rd`. A combination of GL [13] and simple rewrite rules enables us to prove "inverse" properties, e.g. the following theorem "recovers" the destination register from an `asm-add` call:

```
(defthm get-rd-of-asm-add (equal (get-rd (asm-add rs1 rs2 rd)) (n05 rd)))
```

Theorems of this sort are proven for every field (i.e. `funct7`, `rs2`, `rs1`, `funct3`, `rd`, `opcode`, and all `imm` variations) of every RV32I instruction. Thus, even though calls are performed using `get-opcode`, `get-funct3`, and `get-funct7` early within our "step" function, proving the correctness of RV32I instructions does not rely on opening these decoding functions. Similarly, calls within instruction semantic functions are made to some subset of `get-rs1`, `get-rs2`, `get-rd`, and various "get immediate" functions, but these decoding functions are almost always disabled. This approach enables us to readily verify the effects of every RV32I instruction on the `rv32` state. For example, the following theorem determines a priori the state of an `rv32` object with a PC pointing to the beginning of an "add" instruction after one fetch-decode-execute cycle:

```
(defthm rv32-step-asm-add-correctness
 (implies (and (rv32p rv32)                                          ;; rv32 is well-formed
               (< (xpc rv32) *2^32-5*)                               ;; PC within memory bounds
               (not (equal (n05 k) 0))                               ;; dest reg is not x0
               (equal (rm32 (xpc rv32) rv32) (asm-add i j k)))       ;; ADD instruction at PC
          (equal (rv32-step rv32)                                    ;; execute 1 CPU cycle
                 (!xpc (+ (xpc rv32) 4)                              ;; update PC
                       (!rgfi (n05 k)
                              (n32+ (rgfi (n05 i) rv32) (rgfi (n05 j) rv32))  ;; reg[k] <- reg[i] + reg[j]
                              rv32)))))
```

The upshot is that we now have a verified (with respect to a cycle of the RISC-V CPU) encoding / semantic function pair for each RISC-V instruction, the proofs for which are entirely automatic.

Some previous machine models in the operational semantics tradition perform all the decoding at the top-level within the step and instruction semantic functions (e.g. a single semantic function may decode by bit twiddling without dispatching to another function) making theorems about a single cycle dependent entirely on opening a single complicated function. This can hinder verification efforts for

programs whose inputs are abstracted away or not yet known (e.g. free variables representing immediate values). A slight novelty in this model is that we offload all the decoding to a decoding-specific "layer" of functions; explicitly, we call instruction decoding functions within the step function (see code snippet for `rv32-step` above) and the instruction semantic functions. We prove relevant theorems for these decoding functions so that future RV32I program verification efforts are more amenable. It is much easier to prove theorems about pure machine code / bitvectors without the burden of a CPU structure; the ACL2 code for `get-rd-of-asm-add` above is an example of such a theorem. Similarly, it is much easier to prove theorems about a pure CPU structure without having to worry about bit twiddling; for example, proving the theorem `rv32-step-asm-add-correctness` in the previous paragraph does not involve opening any instruction encoding / decoding functions but instead relies on the encoding / decoding inversion rules. Connecting the two layers enables us to prove the desired theorems about the full fetch-decode-execute cycle by reducing to theorems already proven about the individual layers.

Our RV32I model is highly inspired by similar ACL2 work for the Y86 [2], CHERI-Y86 [9], and x86 [6] ISAs. This work is also partially motivated by the recent development of zero-knowledge virtual machines (i.e. virtual machines which enable one party to prove properties about a program trace to another party without revealing certain information, such as the program inputs) for RISC-V programs [8]. One future direction is to develop a verified assembler for RISC-V assembly into machine code. Note that our instruction encoding functions, (e.g. `asm-add`) is very near to an assembly function that might parse a string specifying a RISC-V instruction (e.g. integer addition) and return the output of our encoding function. Furthermore, we are interested in theorems such as `rv32-step-asm-add-correctness` because it may be easier to formalize a more general theorem for an assembler with respect to a CPU cycle when instructions involve free variables, such as in the function call `(asm-add i j k)`. This is in contrast to code proofs, where symbolic execution involving explicit constants can be common. Another direction for future work is to continue modelling other 32- or 64-bit RISC-V extensions. On one hand, our experience in formalizing and verifying the base RV32 instructions involved many repetitive tasks, suggesting future RISC-V extensions to this model and accompanying theorems can be easily synthesized, by way of macros or otherwise. On another hand, our current memory is modelled as a single stobj array, which is manageable for RV32 but not for RV64. We must improve the resource usage of our memory model, perhaps by using abstract stobjs similar to how the bigmem [7, 3] project is implemented, before tackling the RISC-V 64-bit instruction set. While there is much future work to be done, we are optimistic that the application of our model to the formal analysis and verification of assemblers, programs, hardware, virtual machines, and other artifacts will result in more reliable RISC-V infrastructure.

References

[1] *Stobj*. `https://www.cs.utexas.edu/~moore/acl2/manuals/current/manual/?topic=ACL2____STOBJ`. Accessed 2024-04-15.

[2] (2025): *ACL2 Y86 Models*. `https://github.com/acl2/acl2/tree/master/books/models/y86`. Accessed 2025-01-29.

[3] (2025): *Bigmem*. `https://www.cs.utexas.edu/~moore/acl2/manuals/latest/?topic=BIGMEM____BIGMEM`. Accessed 2025-01-29.

[4] Derek Atkins, Arvind, Krste Asanović, Rimas Avižienis, Jacob Bachmeyer, Christopher F. Batten, Allen J. Baum, Abel Bernabeu, Alex Bradbury, Scott Beamer, Hans Boehm, Preston Briggs, Christopher Celio, Chuanhua Chang, David Chisnall, Paul Clayton, Palmer Dabbelt, L Peter Deutsch, Ken Dockser, Paul Donahue, Aaron Durbin, Roger Espasa, Greg Favor, Andy Glew, Shaked Flur, Stefan Freudenberger, Marc

Gauthier, Andy Glew, Jan Gray, Gianluca Guida, Michael Hamburg, John Hauser, John Ingalls, David Horner, Bruce Hoult, Bill Huffman, Alexandre Joannou, Olof Johansson, Ben Keller, David Kruckemyer, Tariq Kurd, Yunsup Lee, Paul Loewenstein, Daniel Lustig, Yatin Manerkar, Luc Maranget, Ben Marshall, Margaret Martonosi, Phil McCoy, Nathan Menhorn, Christoph Müllner, Joseph Myers, Vijayanand Nagarajan, Rishiyur Nikhil, Jonas Oberhauser, Stefan O'Rear, Markku-Juhani O. Saarinen, Albert Ou, John Ousterhout, Daniel Page, David Patterson, Christopher Pulte, Jose Renau, Josh Scheid, Colin Schmidt, Peter Sewell, Susmit Sarkar, Ved Shanbhogue, Brent Spinney, Brendan Sweeney, Michael Taylor, Wesley Terpstra, Matt Thomas, Tommy Thorn, Philipp Tomsich, Caroline Trippel, Ray VanDeWalker, Muralidaran Vijayaraghavan, Megan Wachs, Paul Wamsley, Andrew Waterman, Robert Watson, David Weaver, Derek Williams, Claire Wolf, Andrew Wright, Reinoud Zandijk & Sizhuo Zhang (2024): *The RISC-V Instruction Set Manual Volume I: Unprivileged Architecture*. Technical Report. Available at `https://drive.google.com/file/d/1uviu1nH-tScFfgrovvFCrj7Omv8tFtkp/view?usp=drive_link`.

[5] Robert S. Boyer & J. Strother Moore (2002): *Single-Threaded Objects in ACL2*. In Shriram Krishnamurthi & C. R. Ramakrishnan, editors: *Practical Aspects of Declarative Languages*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 9–27, doi:10.1007/3-540-45587-6_3.

[6] Shilpi Goel (2016): *Formal Verification of Application and System Programs Based on a Validated x86 ISA Model*. Ph.D. thesis, University of Texas at Austin.

[7] Warren A. Hunt Jr. & Matt Kaufmann (2012): *A formal model of a large memory that supports efficient execution*. In Gianpiero Cabodi & Satnam Singh, editors: *Formal Methods in Computer-Aided Design, FMCAD 2012, Cambridge, UK, October 22-25, 2012*, IEEE, pp. 60–67. Available at `https://ieeexplore.ieee.org/document/6462556/`.

[8] Carl Kwan, Quang Dao & Justin Thaler (2024): *Verifying Jolt zkVM Lookup Semantics*. Cryptology ePrint Archive, Paper 2024/1841. Available at `https://eprint.iacr.org/2024/1841`.

[9] Carl Kwan, Yutong Xin & William D. Young (2023): *CHERI Concentrate in ACL2*. In Alessandro Coglio & Sol Swords, editors: Proceedings of the 18th International Workshop on the *ACL2 Theorem Prover and Its Applications*, Austin, TX, USA and online, November 13-14, 2023, *Electronic Proceedings in Theoretical Computer Science* 393, Open Publishing Association, p. 6–10, doi:10.4204/EPTCS.393.2.

[10] Sandip Ray, Warren A. Hunt, John Matthews & J. Strother Moore (2008): *A Mechanical Analysis of Program Verification Strategies*. *Journal of Automated Reasoning* 40, pp. 245–269, doi:10.1007/s10817-008-9098-1.

[11] Sandip Ray & J. Strother Moore (2004): *Proof Styles in Operational Semantics*. In Alan J. Hu & Andrew K. Martin, editors: *Formal Methods in Computer-Aided Design*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 67–81, doi:10.1007/978-3-540-30494-4_6.

[12] RISC-V International (2022): *RISC-V Sees Significant Growth and Technical Progress in 2022 with Billions of RISC-V Cores in Market*. Available at `https://riscv.org/riscv-news/2022/12/risc-v-sees-significant-growth-and-technical-progress-in-2022-with-billions-of-risc-v-cores-in-market/`.

[13] 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.

On Automating Proofs of Multiplier Adder Trees using the RTL Books

Mayank Manjrekar
Austin Design Center
Arm Inc.
mayank.manjrekar2@arm.com

We present an experimental, verified clause processor `ctv-cp` that fits into the framework used at Arm for formal verification of arithmetic hardware designs. This largely automates the ACL2 proof development effort for integer multiplier modules that exist in designs ranging from floating-point division to matrix multiplication.

1 Introduction

Formal verification of multipliers is a difficult problem. At Arm, we have a well-established methodology [4, 3] for verifying arithmetic hardware designs. Verification of a design is a two-step process. First, we model the RTL using the RAC programming language [3], a restricted subset of C++ augmented with AC datatypes [1], and prove it equivalent to the design using an industrial equivalence checker. Second, we use the RAC parser to automatically translate the RAC model into ACL2 and prove that it is correct with respect to a high-level specification; we use mathematical abstractions in the RTL library [5] where, e.g., floating-point operations are specified using rational numbers. Developing the RAC model requires a delicate balance: a higher level of abstraction favors ACL2 proofs but a lower level favors equivalence checks. In this paper, we present an experimental, verified clause processor `ctv-cp` [2] that fits into our framework and largely automates the ACL2 proof development effort for integer multipliers. It allows the RAC model to directly mimic a large portion of the RTL, thereby simplifying model development and facilitating fast equivalence checks.

The design of an integer multiplier may be divided into two parts: the generation and the summation of partial products. Various optimization techniques are employed for performance, but the above partitioning is accurate in principle. Summation of the partial products is done by a *compression tree* circuit that has the largest proportion of the multiplier's area. The compression tree performs a sequence of steps to eventually reduce the number of partial products to two. The two output vectors of the reduction are added together using a carry-propagate adder. Each reduction step is typically implemented using a 3:2 compressor, whose output vectors, *sum* and *carry*, have the following formula: $sum = x \oplus y \oplus z,\ carry = (x \wedge y) \vee (x \wedge z) \vee (y \wedge z)$. Figure 1 shows a bit-matrix representation of a compression tree of a simple 8×8 multiplier; a dot indicates that the corresponding bit may be non-zero.

We also split the verification task along the above separation in the design. We define two separate RAC functions for integer multipliers — `genPP` to generate the partial products and `compress` to mimic the compression tree and the final adder. For the final correctness result, we need to prove that the sum of the partial products generated by `genPP` is equal to the product, and that the `compress` function's summation strategy is correct. In this paper, we focus on automating the proofs of the implementations of the compression tree, i.e., the RAC `compress` function. Note that we verify the corresponding ACL2 definition of `compress`, which is automatically generated by the RAC parser. See examples below for both the RAC and its ACL2 translation for our 8×8 running example, where some code is elided.

Gamboa and Manolios (Eds):
ACL2 Workshop 2025
EPTCS 423, 2025, pp. 51–55, doi:10.4204/EPTCS.423.5

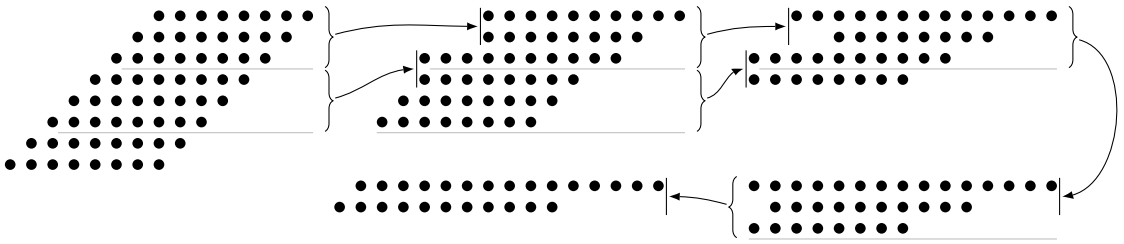

Figure 1: An 8×8 multiplier compression tree

```
// RAC Functions
ui16 compress(ui16 pp0, ui16 pp1, ui16 pp2, ... , ui16 pp7) {
  ui16 l1pp0 = pp0^pp1^pp2;
  ui16 l1pp1 = ((pp0&pp1) | (pp0&pp2) | (pp1&pp2)) << 1;
  ...
  ui16 l4pp0 = l3pp0^l3pp1^l3pp2;
  ui16 l4pp1 = ((l3pp0&l3pp1) | (l3pp0&l3pp2) | (l3pp1&l3pp2)) << 1;
  return l4pp0 + l4pp1; }

ui16 computeProd(ui8 a, ui8 b) {
  array<ui16,8> pp = genPP(a, b);
  return compress(pp[0], pp[1], pp[2], pp[3], pp[4], pp[5], pp[6], pp[7]); }
```

```
;; ACL2 Translation
(defund compress (pp0 pp1 pp2 pp3 pp4 pp5 pp6 pp7)
  (let* ((l1pp0 (setbits 0 16 15 0 (logxor pp0 pp1 pp2)))
         ...
         (l4pp0 (setbits 0 16 15 0 (logxor l3pp0 l3pp1 l3pp2)))
         (l4pp1 (setbits 0 16 15 0
                  (logior (logand l3pp0 l3pp1)
                          (logand l3pp0 l3pp2) (logand l3pp1 l3pp2)))))
    (bits (+ l4pp0 l4pp1) 15 0)))
```

Our new clause processor `ctv-cp` may be invoked as follows to automatically prove the correctness of `compress`.

```
(def-ctv-thm compress-lemma-8x8
  (implies (and (integerp pp0) (integerp pp1) (integerp pp2) (integerp pp3)
                (integerp pp4) (integerp pp5) (integerp pp6) (integerp pp7))
           (equal (compress pp0 pp1 pp2 pp3 pp4 pp5 pp6 pp7)
                  (bits (+ pp0 pp1 pp2 pp3 pp4 pp5 pp6 pp7) 15 0)))
  :expand (compress))
```

2 Algorithm

In principle, the correctness proof of the compression tree may be developed by instantiating Theorem 1 from the RTL books for each 3:2 compressor.

Theorem 1 (Add-3) *If x, y, and z are integers, and* $s = x \oplus y \oplus z$ *and* $c = (x \wedge y) \vee (x \wedge z) \vee (y \wedge z)$*, then* $s + 2c = x + y + z$.

The clause processor `ctv-cp` essentially does this instantiation automatically. The high-level idea is simple; `ctv-cp` works on the LHS and RHS of a goal separately and processes terms on each side into an internal format. It then applies a sequence of normalizing transformations. At the end, if the resulting terms are the same, then the goal is proven.

We describe the algorithm by considering the LHS of the conclusion of `compress-lemma-8x8` — `(compress pp0 pp1 pp2 pp3 pp4 pp5 pp6 pp7)`. First, `ctv-cp` expands all the functions listed in its `:expand` hint, i.e., `compress` in our example. The untranslated body of this function contains a sequence of let-bindings, whose translated version is a nested application of lambda forms:

```
((lambda (l0pp0 pp0 pp1 pp2 ... pp7)
  ...
    ((lambda (l4pp0 l4pp1)
       (bits (binary-+ l4pp0 l4pp1) '15 '0))
     l4pp0 (setbits '0 '16 '15 '0 (binary-logior ... ))) ...)
  (setbits '0 '16 '15 '0 (binary-logxor ... )) pp0 pp1 ... pp7)
```

The clause processor acts on this term by diving into the lambda expressions to reach the inner-most term, `(bits (binary-+ l4pp0 l4pp1) '15 '0)`. As it does so, it also builds a *substitution context* needed to interpret the inner-most term. A *substitution* is an association list mapping symbols to ACL2 terms, and a substitution context is a list of such substitutions. In our example, the first substitution is

```
'((l0pp0 . (setbits '0 '16 '15 '0 (binary-logxor ... )))
  (pp0 . pp0) (pp1 . pp1) ... (pp7 . pp7)).
```

Once the inner-most expression is reached, the bit-width of the expression is inferred (16 in the example), and the expression is parsed into a data structure that represents its bitwise expansion. This data structure is specified in BNF for brevity on the left side below, but is defined using the FTY books [6]. The right side shows the interpretations for such data.

bvfsl := (cons *bvfs bvfsl*)	(cons *a b*) $\mapsto$ (+ (*interp a*) (*interp b*))
| nil	nil $\mapsto$ 0
bvfs := '(*bvf num*)	'(*a n*) $\mapsto$ (ash (*interp a*) *n*)
bvf := *bv*	
| '(:fas *bvf bvf bvf*)	'(:fas *a b c*) $\mapsto$ (logxor (*interp a*) (*interp b*) (*interp c*))
| '(:fac *bvf bvf bvf*)	'(:fac *a b c*) $\mapsto$ (logior (logand (*interp a*) (*interp b*)) (logand (*interp a*) (*interp c*)) (logand (*interp b*) (*interp c*)))
bv := '(:bit *term num*)	'(:bit *a n*) $\mapsto$ (bitn (*interp a*) n)
| '(:v 0)	'(:v 0) $\mapsto$ 0
| '(:v 1)	'(:v 1) $\mapsto$ 1

For the running example, the bitwise expansion of the inner-most term is

```
'(((:bit l4pp0 0) 0) ((:bit l4pp0 1) 1) ... ((:bit l4pp0 15) 15)
  ((:bit l4pp1 0) 0) ((:bit l4pp1 1) 1) ... ((:bit l4pp1 15) 15))
```

and its immediate interpretation (in untranslated form for readability) is

```
(+ (ash (bitn l4pp0 0) 0) (ash (bitn l4pp0 1) 1) ... (ash (bitn l4pp0 15) 15)
   (ash (bitn l4pp1 0) 0) (ash (bitn l4pp1 1) 1) ... (ash (bitn l4pp1 15) 15))
```

`ctv-cp` generates the bitwise expansion by repeatedly calling a function called `get-nth-bit`. When given a term *x* and a bit position *n*, this function outputs a *bvf* form that has the interpretation `(bitn` *x n*`)`. The function `get-nth-bit` knows how to parse some RTL library functions such as `bits`, `setbits`, etc., that appear in code generated by the RAC parser. It can also recognize expressions emerging from

instances of 3:2 compressors and generate *bvf* forms of type `:fas` or `:fac`. Specifically, a term of the form `(logxor` a b c`)` yields `(:fas` a' b' c'`)`, and a term of the form `(logior (logand` a b`) (logand` a c`) (logand` b c`))` gives the output `(:fac` a' b' c'`)`, where a', b' and c' are bvf's obtained by recursively calling `get-nth-bit` on a, b, and c respectively. Note that if `get-nth-bit` fails to parse a term, then it–and consequently `ctv-cp`–aborts with an error.

After parsing, `ctv-cp` applies the following transformations until the substitution context is empty.

1. Match all bvfs of the form `((:fas` a b c`)` k`)` and `((:fac` a b c`)` $k+1$`)`, and replace them with the three bvfs' `(`a k`)`, `(`b k`)`, and `(`c k`)`.
2. Apply the most recent substitution in the context to get a new *bvfsl*.

The first transformation is valid because of the `add-3` lemma. To optimize the matching algorithm, we normalize and sort the *bvfs* terms. The function `get-nth-bit` is again used by the substitution step — substituting `(`x `.` *term*`)` in the *bv* form `(:bit` x l`)` gives `(get-nth-bit` *term* l`)`.

An important detail is that the transformations are justified by lemmas in the RTL books that have `integerp` type constraints; see, e.g., the `add-3` lemma. We defer discharging these hypotheses until the end. All `ctv-cp` functions maintain a list of terms that need to satisfy `integerp`, and syntactic analysis is done to resolve such hypotheses whenever a substitution is made. If the final transformed terms for LHS and RHS match, the clause processor tries to prove these type hypotheses under the original assumptions of the theorem; if it cannot, then it prompts the user to supply any missing assumptions.

3 Observations and Related Work

The largest multipliers that we have used `ctv-cp` on so far at Arm have 64×64-bit Dadda and Wallace compression trees; the runtime is less than 1 second. The automation and speed of `ctv-cp` reduces the ACL2 proof development effort for integer multipliers and facilitates quick equivalence checks because the RAC models can faithfully replicate the RTL. We refrain from doing a formal complexity analysis for `ctv-cp`, but note that its runtime is proportional to the size of the *bvfsl* terms and the number of substitutions in the design. The size of the terms is never larger than the product of the number of the initial partial products and the multiplication size (i.e., 16 for an 8×8-bit multiplier). Thus, we expect `ctv-cp` to scale for the multipliers we deal with at Arm.

An alternative approach for verifying compression trees would be to apply rewriting after the beta-reduction of lambda terms. For efficiency, such an approach would need structure sharing using hash-consing, outside-in rewriting, and optimized algorithms for term matching. Our implementation is simple; it operates on lambda terms and applies the matching algorithm from the inner-most term outwards before applying substitutions; this is equivalent in principle to the alternative approach above, and obviates the need for such nontrivial optimization techniques.

In related work [8, 7], the author develops an efficient, automatic tool, VeSCMul, for end-to-end proofs of a wide variety of multiplier designs in ACL2. A rewriting-based approach is used that employs optimization techniques to avoid costly backchaining. Unfortunately, VeSCMul does not currently work with functions in the RTL books, which are present in the code generated by the RAC parser. Instead of implementing a translator, we developed `ctv-cp` which has a simple implementation, works seamlessly with our existing verification methodology, and has the advantage that it normalizes terms until fixpoint, which is conducive to producing informative messages if any errors are encountered.

In the future, we plan to develop automation for reasoning about the partial product generation step to reduce the verification overhead of obtaining end-to-end correctness proofs for integer multipliers and subsequently, other design units that include them.

References

[1] *Algorithmic C datatypes*. https://github.com/hlslibs/ac_types. Accessed: 2025-04-27.

[2] Mayank Manjrekar: *ctv-cp clause-processor*. https://github.com/acl2/acl2/tree/master/books/workshops/2025/manjrekar. Accessed: 2025-04-27.

[3] 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.

[4] David M. Russinoff, Javier D. Bruguera, Cuong Chau, Mayank Manjrekar, Nicholas Pfister & Harsha Valsaraju (2022): *Formal Verification of a Chained Multiply-Add Design: Combining Theorem Proving and Equivalence Checking*. In: *29th IEEE Symposium on Computer Arithmetic, ARITH 2022, Lyon, France, September 12-14, 2022*, IEEE, pp. 120–126, doi:10.1109/ARITH54963.2022.00030.

[5] David M. Russinoff et al.: *RTL Books*. https://www.cs.utexas.edu/~moore/acl2/manuals/latest/index.html?topic=ACL2____RTL. Accessed: 2025-04-27.

[6] Sol Swords & Jared Davis (2015): *Fix Your Types*. In Matt Kaufmann & David L. Rager, editors: *Proceedings Thirteenth International Workshop on the ACL2 Theorem Prover and Its Applications, Austin, Texas, USA, 1-2 October 2015*, *EPTCS* 192, pp. 3–16, doi:10.4204/EPTCS.192.2.

[7] Mertcan Temel (2022): *Verified Implementation of an Efficient Term-Rewriting Algorithm for Multiplier Verification on ACL2*. In Rob Sumners & Cuong Chau, editors: *Proceedings Seventeenth International Workshop on the ACL2 Theorem Prover and its Applications, Austin, Texas, USA, 26th-27th May 2022*, *EPTCS* 359, pp. 116–133, doi:10.4204/EPTCS.359.11.

[8] Mertcan Temel (2024): *VeSCMul: Verified Implementation of S-C-Rewriting for Multiplier Verification*. In Bernd Finkbeiner & Laura Kovács, editors: *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part I*, *Lecture Notes in Computer Science* 14570, Springer, pp. 340–349, doi:10.1007/978-3-031-57246-3_19.

Extended Abstract: Partial-encapsulate and Its Support for Floating-point Operations in ACL2

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

1 Introduction

The partial-encapsulate[1] macro was introduced in ACL2 Version 8.2 (May, 2019), providing a general way to evaluate constrained functions, thus generalizing trusted (unverified) clause-processors [5]. However, the ACL2 community books [6] of ACL2 Version 8.6 contain only a few applications of this utility. One goal of this extended abstract is to publicize (finally) this powerful utility. We do so by describing how it supports floating-point (FP) computation in ACL2, which addresses our second goal: to augment the very brief discussion of that support in our published treatment of FP computation in ACL2 [4].

This extended abstract is intended to be reasonably self-contained, especially when combined with the supporting materials described below. For much more background on FP computation in ACL2, see its documentation topic for user-level discussion; and for implementation-level comments, see the ACL2 source code, especially file `float-a.lisp` and the comment therein, *Essay on Support for Floating-point (double-float, df) Operations in ACL2*.

FP computations are widely used in the scientific community, and they are generally much faster than computations with rationals. ACL2 supports such computations using *double-floats* (FPs), which are a Lisp[2] datatype typically consisting of double-precision floating-point numbers. But FP operations are awkward to axiomatize. The following Lisp computations show that FP addition is not associative (which is awkward since ACL2 + is axiomatized to be associative) and in Lisp, the `EQUAL` function does not compute equality on numbers.

```
? (setq *read-default-float-format* 'double-float) ; read FPs as double-floats
DOUBLE-FLOAT
? (+ 0.1 (+ 0.2 0.3))
0.6
? (+ (+ 0.1 0.2) 0.3) ; not the same result as above; associativity fails!
0.6000000000000001
? (equal 1 1.0) ; two equal arithmetic values need not satisfy EQUAL
NIL
?
```

A solution might be to add a new FP datatype to the ACL2 logic, but we were loath to complicate ACL2 that way. In particular, although that could explain a result of `nil` for the evaluation of `(equal 1`

[1] Underlined links are to ACL2 documentation topics.
[2] In this paper, "Lisp" refers to Common Lisp [7].

Gamboa and Manolios (Eds):
ACL2 Workshop 2025
EPTCS 423, 2025, pp. 56–59, doi:10.4204/EPTCS.423.6

1.0), it would be at odds with a result of t for the evaluation of (= 1 1.0), since = is defined logically to be EQUAL. The new datatype would also probably complicate ACL2's type reasoning.

Instead, ACL2 models FPs as the rational numbers they *represent*; a rational is *representable* if it is the numeric value of a double-float. By tracking the use of FP expressions much as stobjs [1] are tracked, ACL2 arranges for Lisp FP computations to be performed using Lisp double-floats, even though they are rational operations logically.

Our goal is to illustrate how partial-encapsulate, when combined with redefinition in Lisp allowed by a trust tag [2], can extend the power of ACL2. We illustrate this idea by showing a toy example that supports FP operations. For simplicity, this exposition ignores the stobj-like tracking mentioned above; as a result (and as noted at the end below), this toy implementation is actually unsound! That observation highlights the potential danger of using partial-encapsulate together with redefinition in Lisp. The actual ACL2 implementation of floating-point operations also uses partial-encapsulate but avoids unsoundness by taking great care, including the use of stobj-like tracking mentioned above.

Although our toy example involves floating-point numbers, we expect most user applications would avoid data types not supported by the ACL2 logic (like floating-point). That should make it considerably less complicated to avoid unsoundness than was the case when adding support to ACL2 for FP operations.

2 A Toy Implementation Illustrating FP Support

We describe the example worked out in the supporting materials for this paper, which can be found in the following files in community books directory books/demos/fp/.

- fp.lisp — Certifiable book introducing some FP operations logically
- fp-raw.lsp — Lisp redefinitions supporting FP computation
- fp.acl2 — Certification support for trust tag and dependencies

They define a few functions with "fp" in the name, which correspond to analogous ACL2 built-ins with "df" (for "double-float") in the name instead of "fp". (There are many more df built-ins as well.) Square root and addition functions are introduced logically in fp.lisp using partial-encapsulate but are given executable Lisp definitions in file fp-raw.lsp. To support these, we also introduce a conversion function to-fp and a recognizer function fpp in fp.lisp, as follows. Think of (to-fp x) as choosing a representable rational near x; specifically, it chooses the rational returned by evaluating the expression (float x 0.0D0) in Common Lisp, as discussed further below.

```
(partial-encapsulate ; introduce conversion to representable rationals
 (((constrained-to-fp *) => * :formals (x) :guard (rationalp x)))
 nil ; supporters; see documentation for partial-encapsulate
 (local (defun constrained-to-fp (x) (declare (ignore x)) 0))
 (defthm rationalp-constrained-to-fp
   (rationalp (constrained-to-fp x))
   :rule-classes :type-prescription)
 (defthm constrained-to-fp-idempotent
   (equal (constrained-to-fp (constrained-to-fp x))
          (constrained-to-fp x)))
 ...  ; other exported defthm events omitted here
)
```

```
(defun to-fp (x) ; convert to representable rationals
  (declare (xargs :guard (rationalp x)))
  (constrained-to-fp x))
(defun fpp (x) ; recognizer for representable rationals
  (declare (xargs :guard t))
  (and (rationalp x) (= (to-fp x) x)))
```

A `partial-encapsulate` event represents a corresponding, implicit `encapsulate` event that introduces additional exported theorems. The key requirement is that the axioms exported by that event, including the implicit additional ones, are all provable for some choice of local witnesses for the signature functions. See the documentation topic for partial-encapsulate for more information about that utility, in particular its lack of support for functional instantiation due to unknown constraints.

In the case of `constrained-to-fp`, the implicit constraints (from additional, hidden `defthm` events) include a theorem for each computation result based on the following definition from `fp-raw.lisp`; for example, since `(float 1/3 0.0D0)` computes to an FP with value
6004799503160661/18014398509481984, an implicit axiom is
`(equal (to-fp 1/3) 6004799503160661/18014398509481984)`.

```
(defun to-fp (x)
  (declare (type rational x))
  (float x 0.0D0))
```

Of course, there are in principle infinitely many such implicit axioms. But the implicit `encapsulate` event is a finite object, so we consider only computation results that will be performed, somewhere by someone, using the current version of ACL2. For details, see comments in the `partial-encapsulate` that introduces function symbol `constrained-to-df` in ACL2 source file `float-a.lisp`.

Why don't we instead introduce `to-fp` with `partial-encapsulate` and eliminate the function `constrained-to-fp`? The reason is that the ACL2 rewriter refuses to execute calls of constrained functions (regardless of redefinition in Lisp). This way, ACL2 succeeds, for example, in the proof of `(thm (equal (to-fp 1/4) 1/4))`.

The function `fp-round` is similar to `to-fp`, but these two functions serve different purposes. `To-fp` is intended to be executable. `Fp-round`, which is not executable, logically supports defining FP addition to be the rounded result of exact addition, as specified by IEEE Standard 754 [3]. FP addition is defined as follows in `fp.lisp`.

```
(defun fp+ (x y)
  (declare (xargs :guard (and (fpp x) (fpp y))))
  (fp-round (+ x y)))
```

`Fp+` is redefined in `fp-raw.lsp` as follows. Note that for FPs x and y, the Lisp `+` operation does the requisite rounding.

```
(defun fp+ (x y)
  (declare (type double-float x y))
  (+ x y))
```

For more details see the aforementioned supporting materials, which in particular contain:

- redefinition in Lisp using a trust tag followed by the form `(include-raw "fp-raw.lsp")` in `fp.lisp`, to load `fp-raw.lsp` into Lisp, which redefines functions already defined in ACL2;

- introduction of the FP square root function, `fp-sqrt`, using `partial-encapsulate` for ACL2 and Lisp `sqrt` for execution;
- handling of executable-counterpart (so-called "*1*") functions for redefined functions;
- tests showing that evaluation works, even during proofs; and
- examples demonstrating the need for care when using Lisp redefinition, by proving `nil`.

The soundness issue just above is due to the attempt to traffic in a Lisp datatype (double-float) that is not supported in the ACL2 logic. Comments in `fp.lisp` outline how ACL2 avoids these problems for its df implementation. We expect that most user applications of `partial-encapsulate` can avoid such soundness issues if appropriate care is taken.

Acknowledgments. We thank Warren Hunt for encouraging the implementation of floating-point operations in ACL2 and ForrestHunt, Inc. for supporting that implementation. We also thank the reviewers for helpful comments.

References

[1] Robert S. Boyer & J Strother Moore (2002): *Single-Threaded Objects in ACL2*. In Shriram Krishnamurthi & C. R. Ramakrishnan, editors: *Practical Aspects of Declarative Languages, 4th International Symposium, PADL 2002, Portland, OR, USA, January 19-20, 2002, Proceedings*, *Lecture Notes in Computer Science* 2257, Springer, pp. 9–27, doi:10.1007/3-540-45587-6_3.

[2] Peter C. Dillinger, Matt Kaufmann & Panagiotis Manolios (2007): *Hacking and Extending ACL2*. In Ruben Gamboa, Jun Sawada & John Cowles, editors: *Proceedings Seventh International Workshop on the ACL2 Theorem Prover and its Applications*.

[3] IEEE (2019): *IEEE Standard for Floating-Point Arithmetic*. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, doi:10.1109/IEEESTD.2019.8766229.

[4] Matt Kaufmann & J Strother Moore (2024): *ACL2 Support for Floating-Point Computations*, p. 251–270. Springer Nature Switzerland, doi:10.1007/978-3-031-66676-6_13.

[5] Matt Kaufmann, J Strother Moore, Sandip Ray & Erik Reeber (2009): *Integrating External Deduction Tools with ACL2*. *Journal of Applied Logic* 7(1), pp. 3–25, doi:10.1016/j.jal.2007.07.002.

[6] The ACL2 Community (2024): *The ACL2 Community Books*. `https://github.com/acl2/acl2/tree/master/books`.

[7] Kent Pitman: *The Common Lisp HyperSpec*. See `https://www.lispworks.com/documentation/HyperSpec/Front/`.

Extended Abstract: Mutable Objects with Several Implementations

Matt Kaufmann
University of Texas at Austin (retired), Austin, TX, USA
kaufmann@cs.utexas.edu

Yahya Sohail
University of Texas at Austin, Austin, TX, USA
yahya@yahyasohail.com

Warren A. Hunt, Jr.
University of Texas at Austin, Austin, TX, USA
hunt@cs.utexas.edu

This extended abstract outlines an ACL2 feature, attach-stobj[1], that first appeared in ACL2 Version 8.6 (October, 2024). Familiarity is assumed here with single-threaded objects, or *stobjs* [1] — not only ordinary concrete stobjs but also abstract stobjs [2].

For a worked example that illustrates attach-stobj, see the directory demos/attach-stobj/ in the community books [4], starting with file README.txt in that directory. Performance is addressed in subdirectory mem-test/ of that directory and is discussed in Section 3 below.

1 Background and Acknowledgments

The evolving x86 model [3] (in community books directory projects/x86isa/) currently represents its memory using an abstract stobj that is nested in X86, the abstract stobj it uses to represent its state. Linux has been booted on this model and Linux jobs have been run on it. But for efficient execution for a variety of applications, we wanted the x86 model to be flexible by permitting different memory models to be used with it. This paper describes an enhancement to ACL2 that permits such substitution of memory models without requiring recertification of the book that defines the X86 abstract stobj. We thank Sol Swords for helpful design feedback; ForrestHunt, Inc. for supporting the research reported herein; and the reviewers for helpful feedback on this paper.

2 Overview

The idea is to allow an abstract stobj ST to be defined in a book as an *attachable* stobj, using keyword argument :attachable t as shown below, so that different ACL2 sessions can specify different ways to execute operations on ST without the need to recertify the book that defines ST. In particular, those overriding executions can be available without re-proving the theorems that have been proved about ST.

Let's outline how this works. We start with a book, B_ST, that contains a defabsstobj event introducing the stobj, ST, followed by some theorems.

```
(defabsstobj ST
  ...
```

[1] Underlined links are to ACL2 documentation topics. In particular, the topic for attach-stobj has details not included in this abstract.

Gamboa and Manolios (Eds):
ACL2 Workshop 2025
EPTCS 423, 2025, pp. 60–64, doi:10.4204/EPTCS.423.7

```
  :attachable t)    ; allows execution of ST to be modified; see below
;;; ... some theorems ...
```

Then, one or more other books could look as follows, where IMPL is given the same sequence of :logic functions in its primitives as those of ST.

```
(defabsstobj IMPL ...) ; Might be top-level, but might be from an included book
(attach-stobj ST IMPL) ; IMPL may be attached to a stobj ST, introduced later.
(include-book "B_ST")  ; Define ST using :attachable t, so that execution with
                       ;   ST is performed as specified by IMPL.
```

The relevant notions and required order are as follows, as illustrated above.

- The *implementation* stobj, IMPL, is defined before the attach-stobj event specifying that it will be *attached to* the *attachable* stobj, ST.
- The attach-stobj event precedes the defabsstobj event that defines ST.

The user documentation for attach-stobj provides detailed requirements for its use. The key idea is to replace the :foundation and the :exec fields of the attachable stobj with those of its implementation.

Suppose that IMPL is to be used only as suggested above — that is, it will only serve as an implementation stobj to be attached to some other stobj (for example, ST above). Then the defabsstobj event for IMPL can save space by specifying :non-executable t. That option's sole effect is to prevent the creation of a global IMPL stobj (which however can be created later, if needed, using add-global-stobj). Option :non-executable t is also useful for a stobj if it will only be used as a child of a superior stobj or as a local stobj.

For the motivating application described in the preceding section, ST and IMPL represent memories within a superior X86 stobj; see nested-stobjs. Attach-stobj works as expected when attaching to the child stobj (whether concrete or abstract), essentially as follows.

```
(defabsstobj IMPL ...) ; implementation memory stobj
(attach-stobj ST IMPL) ; IMPL to be attached later to ST
(defabsstobj ST        ; attachable memory stobj
  ...
  :attachable t)
(defabsstobj X86 ...)  ; includes ST as a child, which executes as IMPL
```

As the new capability was being designed, it was considered to support introduction of an attachable stobj without :exec fields for its primitives. But that would have required developing a semantics for such "incomplete" abstract stobj definitions as well as modifying the checks. In particular, what should be done about the correspondence function and theorems? There were also problems involving signatures of exported functions when omitting :exec fields. It thus seemed reasonable to make the usual requirements for abstract stobjs even when they are attachable, which pertain even to :exec fields that might not ultimately be used in execution. One can view local witnesses in encapsulate events as providing a sort of precedent.

3 Performance

This section illustrates how attach-stobj can provide substantial performance benefits without incurring extra proof work. It summarizes results reported in directory demos/attach-stobj/mem-test/

of the community books [4], in particular its file `README.txt`. We start with the following table, which gives runtime and physical memory used in the six runs described below.

Memory	Benchmark	Time (secs)	Size (bytes)
symmetric	low	2.75	2000085072
symmetric	high	2.75	2000085072
asymmetric	low	0.00	6663495760
asymmetric	high	87.91	6666641488
attached	low	0.00	6899818576
attached	high	89.04	6902964304

The rows are based on doing 100,000 writes of byte value 1 to random addresses in a range of 2^{30} contiguous addresses. Those addresses start at address 0 for "low" writes and at $6 * 2^{30}$ for "high" writes. The writes are done after loading memory models using the following ACL2 commands.

- *symmetric*:
  ```
  (include-book "centaur/bigmems/bigmem/bigmem" :dir :system)
  ```
- *asymmetric*:
  ```
  (include-book "centaur/bigmems/bigmem-asymmetric/bigmem-asymmetric" :dir :system)
  ```
- *attached*:
  ```
  (include-book "centaur/bigmems/bigmem-asymmetric/bigmem-asymmetric" :dir :system)
  (attach-stobj bigmem::mem bigmem-asymmetric::mem)
  (include-book "centaur/bigmems/bigmem/bigmem" :dir :system)
  ```

In all cases, we see that the *symmetric* memory model performs best of the three when writing to "high" memory, while the other two memory models perform best when writing to "low" memory. It can thus be beneficial to choose different memory models for different applications.

Naive implementations would simply create the *symmetric* and *asymmetric* models and prove desired theorems about each. But with `attach-stobj`, we need only prove theorems about the *symmetric* model: with `attach-stobj` one can *attach* the *asymmetric* model to the *symmetric* model when one wants the performance provided by the *asymmetric* model.

This use of `attach-stobj` has benefit beyond avoiding the duplication of proofs. Books that include the *symmetric* model can be used with either model, depending on whether or not the *asymmetric* model is first included and then attached to the *symmetric* model, as shown in the three commands displayed above for the *attached* usage. Without the availability of `attach-stobj`, one would need to develop two such books: one that includes the *symmetric* model and one that includes the *asymmetric* model.

Note that the performance penalties are minor for using *attached* instead of *asymmetric*. This is a small price to pay for the benefits described above.

4 Implementation Notes

The basic implementation idea for attachable stobjs is reasonably straightforward. The `attach-stobj` event populates a table, `attach-stobj-table`: it associates an attachable stobj name, which must not yet be defined, with an implementation stobj name, which must already be defined. Then when the attachable stobj is later defined, its corresponding implementation stobj is found by looking in the table — recursively, since the value may itself have an attachment.

```
(defun attached-stobj (st wrld top)
; Top is t for a top-level call, nil otherwise.
  (let ((st2 (cdr (assoc-eq st (table-alist 'attach-stobj-table wrld)))))
    (cond (st2 (attached-stobj st2 wrld nil))
          (top nil)
          (t st))))
```

The main ACL2 source function for implementing `defabsstobj`, which is `defabsstobj-fn1`, calls itself one time recursively when `:attachable t` is supplied, essentially by replacing the `:exec` (execution) fields and `:foundation` of the attachable stobj with those of the implementation stobj. But first it checks that those two abstract stobjs (attachable and implementation) have the same sequence of `:logic` fields. See the source code for the many details omitted here, in particular, the definition of the function `defabsstobj-fn1` mentioned above and the comment entitled "Essay on Attachable Stobjs".

The trickiest part is to install the proper code for execution. When including a certified book that defines an abstract stobj, code for that stobj's primitives is normally provided by the book's compiled file, based on the `:exec` fields of the `defabsstobj` event. But when the stobj is attachable and an attachment is provided, the `:exec` fields of the implementation stobj need to be used instead. This presents a challenge, especially since abstract stobj primitives are macros. Consider for example a book `B_ST.lisp` that contains the following events.

```
(defabsstobj ST
  ...
  :exports (... (p :logic p$a :exec p$c) ...)
  :attachable t)
(defun f (ST) (declare (xargs :stobjs ST)) ... (p ...) ...)
```

Now suppose we attach a stobj `IMPL` to `ST` before including that book.

```
(defabsstobj IMPL
  ...
  :exports (... (p{impl} :logic p$a :exec p{impl}$c) ..))
(attach-stobj ST IMPL)
(include-book "B_ST") ; defines ST and (f ST); see above
```

A naive implementation would load compiled code from the book `B_ST` when it is included; and since `p` is a macro, the compiled code for `f` would be produced by macroexpanding calls of `p` by calling `p$c`. But instead, those calls of `p$c` should instead be calls of the corresponding `:exec` field from `IMPL`, namely, `p{impl}$c`.

This problem is addressed using two globals in the logical world, `ext-gens` and `ext-gen-barriers`, that track functions like `f` for which compiled code from a book should be ignored, so that primitives of an attached stobj are invoked using their attachments. This will generally cause a function like `f` to be compiled when its ACL2 definitional event is encountered during `include-book`. Details are beyond the scope of this extended abstract. However, that and other implementation issues are covered in the Essay mentioned above. The community books directory `system/tests/attachable-stobjs/` has examples that test aspects of attachable stobjs, including some of the trickier aspects of execution that involve them.

We conclude by noting that execution with attachable stobjs is efficient, in that `attach-stobj` introduces no indirection. The trade-off is that compilation is performed at `include-book` time when existing compiled code is avoided, as discussed above.

References

[1] Robert S. Boyer & J Strother Moore (2002): *Single-Threaded Objects in ACL2*. In Shriram Krishnamurthi & C. R. Ramakrishnan, editors: *Practical Aspects of Declarative Languages, 4th International Symposium, PADL 2002, Portland, OR, USA, January 19-20, 2002, Proceedings*, *Lecture Notes in Computer Science* 2257, Springer, pp. 9–27, doi:10.1007/3-540-45587-6_3.

[2] Shilpi Goel, Warren A. Hunt, Jr. & Matt Kaufmann (2013): *Abstract Stobjs and Their Application to ISA Modeling*. *Electronic Proceedings in Theoretical Computer Science* 114, p. 54–69, doi:10.4204/eptcs.114.5.

[3] Shilpi Goel, Warren A. Hunt, Jr. & Matt Kaufmann (2014): *Simulation and Formal Verification of x86 Machine-Code Programs that make System Calls*. In K. Claessen & V. Kuncak, editors: *FMCAD'14: Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design*, EPFL, Switzerland, pp. 91–98, doi:10.1109/FMCAD.2014.6987600.

[4] The ACL2 Community (2024): *The ACL2 Community Books*. https://github.com/acl2/acl2/tree/master/books.

A Formalization of the Yul Language and Some Verified Yul Code Transformations

Alessandro Coglio Eric McCarthy
Kestrel Institute https://kestrel.edu

Yul is an intermediate language used in the compilation of the Solidity programming language for Ethereum smart contracts. The compiler applies customizable sequences of transformations to Yul code. To help ensure the correctness of these transformations and their sequencing, we used the ACL2 theorem prover to develop a formalization of the syntax and semantics of Yul, proofs relating static and dynamic semantics, a formalization of some Yul code transformations, and correctness proofs for these transformations.

1 Introduction

Solidity [21, 18] is a programming language for writing smart contracts for the Ethereum blockchain [9]. Solidity is compiled to EVM (Ethereum Virtual Machine) bytecode [26], which is directly executed by transactions on the blockchain. The Solidity compiler includes the preferred option to translate Solidity to EVM bytecode via the intermediate language Yul [20] (see Figure 1): first, Solidity is turned into Yul with a relatively simple translation; next, the Yul code undergoes several optimizing transformations; finally, the optimized Yul code is turned into EVM bytecode with another relatively simple translation. The rationale is to move most of the compilation complexity into the Yul code transformations, which eases the task because Yul is simpler than Solidity and more structured than EVM bytecode. Yul is also used to write inline assembly in Solidity, i.e. to embed EVM bytecode (with Yul syntax) directly in the Solidity code, which is sometimes necessary in Ethereum smart contracts.

Yul is designed to be usable as an intermediate language to compile other front ends than Solidity to other back ends than EVM bytecode. In line with this aspiration, Yul consists of a core ('pure Yul' or 'generic Yul') independent from front and back ends, which is extensible with dialects tailored to specific front and back ends. Currently only the EVM dialect is defined, for compiling Solidity to EVM bytecode. A Yul dialect extends the Yul core with specific types and operations.

A few tens of Yul transformations have been defined and implemented [19]. Some are dialect-independent, while others are EVM-dialect-specific. Some transformations assume that others have already taken place, i.e. they expect the code to be in a certain form, which the previous transformations

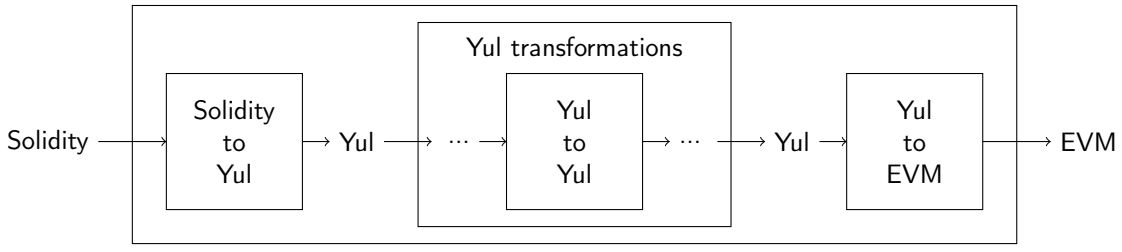

Figure 1: Solidity Compiler with Yul Transformations

Gamboa and Manolios (Eds):
ACL2 Workshop 2025
EPTCS 423, 2025, pp. 65–83, doi:10.4204/EPTCS.423.8

produce. Some transformations, or sequences of transformations, may be iterated, i.e. applied multiple times until either nothing changes or an iteration limit is reached. The Solidity compiler uses a default sequence of transformations, which can be overridden by the user.

It is critical that Yul transformations and their sequencing are correct. Each transformation must be applied to code of the expected form, as produced by the preceding transformations, and must produce code semantically equivalent to the input code. This paper reports on our preliminary work towards addressing these problems, using the ACL2 theorem prover [12]. We developed a formalization of the syntax and semantics of Yul, which covers the generic core and a small portion of the EVM dialect; we proved some properties of the formalization, most notably that the static semantic checks rule out dynamic semantic errors. We formalized some Yul transformations, and verified that they preserve both static and dynamic semantics. We formalized some restrictions on Yul code that are expected by some transformations, and verified that they are preserved by some transformations. Although we have only scratched the surface of verifying transformations and their sequencing, we believe that our work shows the feasibility of the approach. Our ACL2 library for Yul, which contains our development, is available at [25, `[books]/kestrel/yul`] and documented at [24, `yul`].

After providing some background on Yul in Section 2, we describe our formalization of Yul in Section 3, and our work on verified transformations in Section 4. Related work is surveyed in Section 5, while future work is discussed in Section 6. Some closing remarks are given in Section 7.

Background on Solidity and the EVM can be found in a variety of sources, starting with the Ethereum web site [9]. However, knowledge of Solidity and the EVM is not required to read this paper.

2 Background on Yul

Yul is a statically typed, block-structured, imperative language. Statements consist of function definitions, variable declarations and assignments, conditionals, loops, control transfers, expressions (for side effects), and nested blocks. Expressions consist of literals, variables, and function calls. A function takes zero or more inputs and returns zero or more outputs; if it returns no outputs, it is only used for side effects, as a statement.

The Yul core has no types, and no syntax to define types. The EVM dialect has a single type `u256`, consisting of 256-bit unsigned integers. If a type is omitted (e.g. in a variable declaration), it defaults to a type specified by the dialect; the EVM dialect necessarily defaults to `u256`, which is the only type.

The Yul core includes boolean literals (`true` and `false`), numeric literals in decimal and hexadecimal base (e.g. `64738` and `0xff0012`), and certain forms of string literals (e.g. `"abc"` and `hex"90a4"`). The EVM dialect defines the meaning of literals as `u256` values: boolean literals denote 0 and 1, numeric literals denote the obvious, and string literals yield byte sequences interpreted as integers in base 256. It is a static error if a literal denotes 2^{256} or more, e.g. if a string literal yields more than 32 bytes.

Variables are declared via the `let` keyword, with or without a type (see above), and with or without an initializing expression; if the latter is missing, the variable is initialized to `0`. In the EVM dialect, `let x` declares a variable `x` of type `u256` initialized to `0`, while `let y := x` declares a variable `y` of type `u256` initialized to the current value of `x`. A variable assignment is like a variable declaration without `let`, e.g. `x := 17` assigns `17` to `x`. A call of a function that returns two or more outputs can be used to initialize, or to assign to, multiple variables, e.g. `let a, b := f(...)` initializes `a` and `b` to the first and second result of `f`. If multiple variables are declared without initializing expressions, they are all initialized to `0`, as in the case of a single variable, e.g. `let a, b`.

A function definition returns results by assigning them to its output variables, which are declared as

part of the function definition, along with the input variables. For example, in a function definition of the form `function f(x, y) -> a, b { ... a := ... b := ... }`, the input variables are `x` and `y`, and the output variables are `a` and `b`. If a function terminates execution without assigning a value to some output variables, the corresponding result is `0`. The `leave` statement can be used to return from a function, whose execution otherwise terminates at the end of its body block.

The Yul core has no built-in functions. The EVM dialect provides several tens of built-in functions, corresponding to EVM bytecode instructions. For example, the function `add`, which takes two `u256` inputs and returns their `u256` sum (modulo 2^{256}), corresponds to the EVM bytecode instruction `ADD`. An assignment like `z := add(x, y)` in inline assembly represents, and is translated to (as the last compilation step), an `ADD` instruction in EVM bytecode. Although the EVM is stack-based, Yul is essentially register-based (where the registers are the variables); the rationale is to facilitate understanding and manipulation of Yul code, while keeping the translation to EVM bytecode still relatively simple.

`for` loops are structurally similar to C and Java: there is an initialization block, a test expression, an update block, and a body block; the `break` and `continue` statements can be used to break out of a loop or to skip the rest of an iteration. `if` conditionals have a 'then' branch and no 'else' branch. `switch` conditionals have `case` branches based on literals, and optional `default` branches. There are no go-to statements; the rationale for this, and for having structured control flow, is to facilitate understanding and manipulation of Yul code.

Blocks are delimited by curly braces, i.e. `{ ... }`, as in many other languages. Statements are not terminated by semicolons; note that there are no infix operators, only function calls, which makes parsing easier. There are line comments `// ...` and block comments `/* ... */`, as in many other languages.

The Yul documentation [20] includes a grammar, as is customary, and a semi-formal semantics, which is much less customary. The latter is an evaluation function over the Yul syntactic constructs, written in a mix of mathematics, pseudo-code, and English prose.

3 Yul Formalization

Our formalization covers the Yul core and a small portion of the EVM dialect. Extending it to cover the whole EVM dialect, and generalizing it to accommodate other dialects, are both future work.

3.1 Abstract Syntax

The abstract syntax is the fulcrum of our development: concrete syntax abstracts to it; static and dynamic semantics are defined on it; and transformations manipulate it. For defining static and dynamic semantics, the abstract syntax could abstract away all the concrete syntax information that does not affect said semantics. But for defining transformations, it is beneficial to retain enough concrete syntax information to reduce incidental differences between code before and after transformations, to facilitate inspection and debugging; however, retaining excessive concrete syntax information may add complexity without significant additional benefit. In formalizing the abstract syntax, we tried to strike the right balance: we keep all the syntactic details of literals, but we drop whitespace and comments.

The abstract syntax is formalized as a collection of algebraic data types, using the fixtype library [23] [24, `fty`]. For example, expressions are formalized as

```
(fty::deftagsum expression
  (:path ((get path)))
  (:literal ((get literal)))
  (:funcall ((get funcall)))
```

```
  :pred expressionp)
```

i.e. an expression is either a path, or a literal, or a function call; a path is a sequence of identifiers separated by dots, which are used as variable names.[1] As another example, statements are formalized as

```
(fty::deftagsum statement
  (:block ((get block)))
  (:variable-single ((name identifier) (init expression-option)))
  (:variable-multi ((names identifier-list) (init funcall-optionp)))
  (:assign-single ((target path) (value expression)))
  (:assign-multi ((targets path-list) (value funcall)))
  (:funcall ((get funcall)))
  (:if ((test expression) (body block)))
  (:switch ((target expression) (cases swcase-list) (default block-option)))
  (:for ((init block) (test expression) (update block) (body block)))
  (:break ())
  (:continue ())
  (:leave ())
  (:fundef ((get fundef)))
  :pred statementp)
```

i.e. a statement is either a block, or a (single or multiple) variable declaration, or a (single or multiple) variable assignment, or a function call, or an (`if` or `switch`) conditional, or a (`for`) loop, or a (`break` or `continue` or `leave`) control transfer, or a function definition. Overall, the definition of the abstract syntax is unremarkable, directly derived from the concrete syntax.

3.2 Concrete Syntax

To formalize the concrete syntax, we developed an ABNF [7, 15] grammar of Yul, as a straightforward transcription of the grammar in [20, 18].[2] For example, the ABNF grammar rule for expressions is

```
expression = path / literal / function-call
```

and the ABNF grammar rule for statements is

```
statement = block / variable-declaration / assignment / function-call
          / if-statement / switch-statement / for-statement
          / %s"leave" / %s"break" / %s"continue" / function-definition
```

which also show the correspondence with the examples in Section 3.1.[3] The verified ABNF grammar parser [2] [24, `abnf::grammar-parser`] turns the ABNF grammar of Yul into an ACL2 representation with formal semantics, according to the formalization of the ABNF notation [2] [24, `abnf::notation`].

As is customary in programming languages, the grammar consists of a lexical sub-grammar, which specifies how sequences of characters are organized into lexemes (i.e. tokens, whitespace, and comments), and a syntactic sub-grammar, which specifies how tokens (after discarding whitespace and comments) are organized into expressions, statements, and related constructs. As is also customary, the lexical sub-grammar is further constrained by taking the longest possible lexeme at each point (e.g. `xy` is a single lexeme, not two lexemes `x` and `y`), and by the fact that keywords are not identifiers. A complete formalization of the concrete syntax should include these restrictions, but this is future work.

[1] The motivation for using paths as variable names seems to be that Yul variables may represent nested fields in Solidity.

[2] As explained in [24, `yul::concrete-syntax`], [20, 18] contains an old grammar and a new grammar, both of which we have transcribed to ABNF. This paper focuses on the new grammar, for which we have also developed a parser.

[3] While the abstract syntax of statements has different cases for single and multiple variable declarations and assignments, the grammar makes that distinction in the rules for `variable-declaration` and `assignment` (not shown here).

We developed an (unverified) executable parser of Yul in ACL2. The lexer is partially generated, via some preliminary ABNF parser generation tools [24, `abnf::defdefparse`]; the rest is handwritten, closely following the lexical grammar. The parser proper is handwritten, closely following the syntactic grammar, according to a recursive descent strategy.

3.3 Static Semantics

The static semantics consists of efficiently checkable restrictions on the syntax, informally stated in [20]. An example is that a function must be called with the right number of arguments. The Solidity compiler must: enforce these restrictions on inline assembly; translate Solidity code to Yul code that satisfies these restrictions; and transform Yul code preserving these restrictions. The static semantics is formalized as executable ACL2 functions that check these restrictions recursively on the abstract syntax.

The Yul scoping rules involve the notions of visibility and accessibility, which differ between functions and variables. They are explained below, using the following code as example:

```
{ // block 1:
  let x
  function f () { // block 2:
    function h () { ... }
    let y
    { // block 3:
      let z
    }
  }
  function g () { // block 4:
    let y
    function h () { ... }
  }
}
```

A function is both visible and accessible in the whole block where its definition occurs, even before the definition, and including all the nested blocks. In the example above: `f` and `g` are visible and accessible everywhere in block 1 (including blocks 2, 3, and 4), but not outside block 1; the `h` defined in `f` is visible and accessible everywhere in block 2 (including block 3), but not outside block 2 (e.g. not in block 4); and so on. A function definition is disallowed if the function name is already visible and accessible, e.g. no function `g` can be declared in `f`. The `h` defined in `f` is distinct from the `h` defined in `g`.

A variable is visible from just after its declaration to the end of the block where it occurs, including all the nested blocks; the variable is accessible in the same portion of the block, except in nested functions. In the example above: `x` is visible in the portion of block 1 just after its declaration, including blocks 2, 3, and 4, but it is not accessible in blocks 2, 3, and 4; the `y` declared in `f` is visible in the portion of block 2 just after its declaration, including block 3, and it is also accessible in block 3; and so on. A variable declaration is disallowed if the variable name is already visible, regardless of whether it is also accessible. The `y` declared in `f` is distinct from the `y` declared in `g`.

Visibility means lexical scoping, i.e. which names can be seen from where, while accessibility means the ability to reference those seen names. When a function is called, a fresh variable area is created, without the ability to reference the variables of the caller: this is why accessibility of variables stops at function boundaries, and why the notions of visibility and accessibility differ for variables. For functions, visibility and accessibility coincide.

Our formalized static semantics checks the above scoping rules, using symbol tables for variables and functions. Since neither the Yul core nor the EVM dialect have syntax for types, symbol tables for variables are just finite sets of variable names (all of which have the same type), while symbol tables for functions are finite maps from function names to function "types", where the latter are pairs (n,m) where n is the number of inputs and m is the number of outputs. A function must be called with n arguments; the call must be a statement if $m = 0$ (for side effects), or used to initialize or assign m variables if $m \neq 0$.

Restrictions on where `break`, `continue`, and `leave` may occur are enforced by calculating and checking the possible ways in which statements and blocks may terminate. There are four possible ways, called 'modes' (also used in the dynamic semantics; see Section 3.4): three modes corresponding to those three statements, and one mode corresponding to 'regular' termination.

Most static semantic checks are dialect-independent, except that literals are interpreted as denoting `u256` values, which are thus checked to be below 2^{256}; this is EVM-dialect-specific. Our static semantics provides the option to initialize the function symbol table with the types of the built-in functions of the EVM dialect, so that Yul code in the EVM dialect can be properly checked.

3.4 Dynamic Semantics

The dynamic semantics is formalized as a defensive big-step executable interpreter of the abstract syntax. Each call of an ACL2 function of the interpreter attempts to execute its input abstract syntax construct completely, recursively executing the sub-constructs. Since the execution of certain constructs may not terminate, the ACL2 functions take, as additional input, an artificial counter that limits the depth of the mutual recursion: the counter is decremented by one at each recursive call, and used as measure, making the termination proof straightforward. The interpreter is defensive in the sense that it checks the necessary safety conditions, e.g. that each function is called with the right number of arguments, without relying on the static semantics (see Section 3.5 for the relation between static and dynamic semantics).

This approach matches the semi-formal semantics in [20], which is also a big-step interpreter. Besides the syntactic construct (expression, statement, etc.), the interpreter takes as input, and returns as output, a global state G, which is dialect-specific, and a local state L, which is dialect-independent; the interpreter also returns one of the four termination modes described in Section 3.3. L is the state of the local variables. In the EVM dialect, G consists of various areas of memory, and provides read access to some blockchain state (e.g. current block number). Our ACL2 interpreter has the same structure.

The ACL2 function to execute statements is

```
(define exec-statement
  ((stmt statementp) (cstate cstatep) (funenv funenvp) (limit natp))
  :returns (outcome soutcome-resultp)
  (b* (((when (zp limit)) ...)) ; return limit error
    (statement-case stmt
                    :block (exec-block stmt.get cstate funenv (1- limit))
                    :leave (make-soutcome :cstate cstate :mode (mode-leave))
                    ...)) ; handle the other kinds of statement
  :measure (nfix limit))
```

where:

- `stmt` is the statement, which is handled by cases (see the type definition in Section 3.1).
- `cstate` is a computation state, which wraps a finite map from variable names to 256-bit unsigned integers, modeling the local state L, tailored to the EVM dialect because values have type `u256`:

```
(fty::defprod cstate
  ((local lstate)) ; finite map from identifiers to values
```

```
    :pred cstatep)
```

We do not yet model the global state G (which is complex), but the reason to wrap the type `lstate` into `cstate` is to accommodate the future addition of a `(global gstate)` component. The local state is a flat map, not a stack of maps corresponding to scopes, because each called function starts a new local state (consisting of the function's input and output variables), and because nested block scopes cannot shadow variables.

- `funenv` is a function environment, i.e. a stack of finite maps from function names to function information consisting of inputs, outputs, and body:

```
(fty::defprod funinfo
  ((inputs identifier-list)
   (outputs identifier-list)
   (body block))
  :pred funinfop)
```

  Unlike the local state, the function environment is a stack because of the different scoping rules of functions compared to variables: the stack corresponds to the lexical scoping of functions; when a function is called, the stack is trimmed down to the scope of that function. With reference to the example code in Section 3.3: when executing the `h` defined in `g`, the function environment contains a scope for block 1 with `f` and `g`, a scope for block 4 with `h`, and a scope for the body of `h`; if `h` calls `f`, the two top scopes are popped, leaving only the one for block 1, because `f` can only access the functions in that scope.
- `limit` is the artificial counter, which `exec-statement` tests with `zp` as first thing, returning an error value indicating that the limit is exhausted if that is the case. This `limit` is decremented at each recursive call, e.g. in the call of `exec-block`, and used as measure for the mutual recursion.
- `outcome` is either an error value or a statement outcome of type

```
(fty::defprod soutcome
  ((cstate cstate)
   (mode mode))
  :pred soutcomep)
```

  which consists of a possibly updated computation state and a mode of termination. Inspired by the `Result` type in Rust, the type `soutcome-result` extends `soutcome` with error values.
- A block statement is executed by executing the block with a separate function `exec-block`, which extends the function environment with a new scope, executes the statements in the block, and then pops the function environment and reduces the local state to the variables before the block (the function `exec-block` is not shown here).
- The execution of a `leave` statement returns the `leave` termination mode without changing the computation state. That termination mode is propagated upwards, and treated the same as regular termination by the ACL2 function `exec-function` that executes Yul functions.
- The code to execute the other kinds of statements is not shown.

The execution of expressions returns an error value or an expression outcome of type

```
(fty::defprod eoutcome
  ((cstate cstate)
   (values value-list))
  :pred eoutcomep)
```

which is analogous to statement outcomes, but with a list of zero or more values instead of a termination mode. Although function calls, and thus expressions, have no side effects on the local state, because each function has its own local state, our ACL2 interpreter accommodates the extension with side effects on the global state, since expression outcomes include a possibly updated computation state.

Besides returning an error value when the artificial limit is exhausted, our ACL2 interpreter returns an error value when a defensive check fails, e.g. a referenced variable or function is not accessible, a `break` is executed outside a loop body, a function is given the wrong number of arguments, etc.

Except for using values of type `u256`, the current interpreter has no support for the EVM dialect. Adding support involves modeling the global state G and modeling the EVM built-in functions via ACL2 code that manipulates the global state. Adding this support is future work.

3.5 Static Soundness

We proved the soundness of the static semantics with respect to the dynamic semantics: if the checks of the static semantics are satisfied, the dynamic semantics never returns an error value, except when the artificial limit is exhausted. In other words, the defensive checks of the dynamic semantics are guaranteed to succeed if the checks of the static semantics succeed. This kind of property provides a major validation of the design and formalization of a programming language. The converse property, i.e. static completeness, namely that if the dynamic semantics never returns error values (except for exhausting the artificial limit) then the static semantics succeeds, cannot hold for any decidable static semantics, because it is undecidable whether the dynamic semantics returns error values.[4]

This formulation of static soundness relies on the fact that, in the Yul core covered by our dynamic semantics (see Section 3.4), the only error values are for the exhaustion of the artificial limit and for defensive checks also checked by the static semantics. If some dialect-specific built-in function can fail in ways not detectable by the static semantics (e.g. division by zero), then static soundness should be reformulated to rule out only the errors detectable by the static semantics.

The static semantics involves function and variable symbol tables, while the dynamic semantics involves function environments and computation states. To formulate static soundness, those are related as follows: a function environment abstracts to a function symbol table, merging the scopes and only keeping the numbers of inputs and outputs of each function; and a computation state abstracts to a variable symbol table, keeping only the variables in the domain of the local state map.

There is a static soundness theorem for each mutually recursive ACL2 function, proved by induction on the mutual recursion. The theorem for expression execution is

```
(defthm exec-expression-static-soundness
  (b* ((results (check-safe-expression
                 expr (cstate-to-vars cstate) (funenv-to-funtable funenv)))
       (outcome (exec-expression expr cstate funenv limit)))
    (implies (and (funenv-safep funenv)
                  (not (reserrp results))
                  (not (reserr-limitp outcome)))
             (and (not (reserrp outcome))
                  (equal (cstate-to-vars (eoutcome->cstate outcome))
                         (cstate-to-vars cstate))
                  (equal (len (eoutcome->values outcome))
                         results)))))
```

[4] For instance, in a conditional statement `if E B`, where `E` is an expression and `B` is a block, `B` may have a static error that never causes a dynamic error due to `E` being always false. In general, it may be possible to prove a static completeness property with respect to an extended dynamic semantics that nondeterministically chooses among all possible branches regardless of the actual values of the tests that control branching, as done in [1]. In `if E B`, after evaluating `E`, that extended dynamic semantics would nondeterministically either execute `B` or skip it, regardless of the value of `E`. As in [1], this hypothetical static completeness property for Yul would show that the checks of the static semantics are, in a sense, the most liberal possible.

where:

- `cstate-to-vars` abstracts a computation state to a variable symbol table.
- `funenv-to-funtable` abstracts a function environment to a function symbol table.
- `check-safe-expression`, from the static semantics, checks the safety of the expression `expr`, returning the number of results if successful, or an error value otherwise.
- `exec-expression`, from the dynamic semantics, executes the expression `expr`, returning an expression outcome, described in Section 3.4.
- The theorem assumes that: the functions in the function environment pass the checks of the static semantics, formalized by `funenv-safep`; `results` is not an error value (recognized by `reserrp`), i.e. `check-safe-expression` succeeds, and `results` is the number of results of `expr`; the execution of `expr` does not exhaust the artificial limit, where `reserr-limitp` recognizes error values that come from exhausting the limit.
- The theorem concludes that: the execution of `expr` does not return an error value; the new computation state abstracts to the same variable symbol table as the old computation state; the actual number of values in the outcome coincides with the statically computed one.

The theorems for the other `exec-...` functions are similar; when statement outcomes are involved instead of expression outcomes, the conclusion about the number of values instead says that the actual termination mode is an element of the set of possible modes calculated by the static semantics.

In these theorems, the `funenv-safep` hypothesis is critical to establish the static safety hypotheses for the code of each called Yul function, which is obtained from the function environment. As the function environment is extended, the preservation of its safety relies on the static safety hypotheses for the code containing the Yul function definitions added to the environment. When the function environment is trimmed, its safety is preserved because it applies element-wise.

The inductively proved theorems about the `exec-...` functions are preceded by, and rely on, the proofs of several theorems about `funenv-safep`, `funenv-to-funtable`, and other ACL2 functions. Overall, the proofs are not conceptually difficult, but involve a bit of work.

4 Yul Transformations

While the formalization of Yul described in Section 3 has value on its own, our primary motivation for developing it was to support the verification of Yul code transformations in the Solidity compiler. Our work on transformations is fairly preliminary, yet illustrative.

4.1 Approach

Since the Solidity compiler is written in C++ [17], verifying the implementation of Yul transformations is a daunting proposition. It is more feasible to generate, each time a Yul transformation is run, a proof of the correctness of the new code with respect to the old code; a verifying (not verified) compiler approach.

Extending the Solidity compiler to generate such proofs is impractical, due to its complexity and ownership. A more viable approach is to (1) replicate the Yul transformations in ACL2, (2) verify correctness properties of the replicated transformations, and (3) validate the replicated transformations by checking that they are consistent with the transformations in the compiler. Performing the third step every time a transformation is run, and instantiating the general theorems of the second step to the run, achieves the same goal as a proof-generating extension of the compiler, but without modifying the compiler; in [5], we introduced the term ‘detached proof-generating extension’ for this approach.

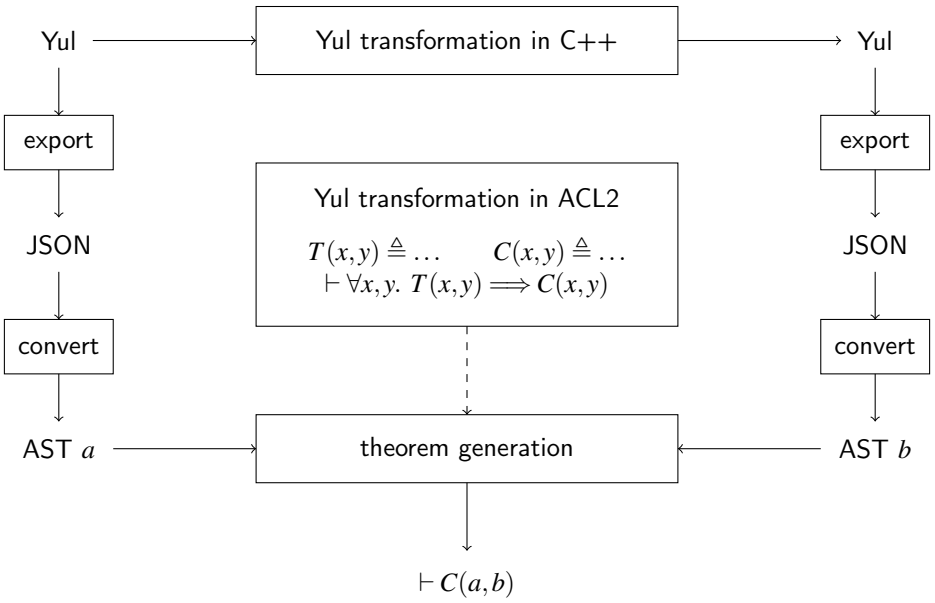

Figure 2: Proof Generation Approach for Yul Transformations

Figure 2 visualizes the approach. The top box is any one of the Yul transformations in the Solidity compiler. The middle box is the replicated transformation in ACL2, formalized as a predicate $T(x,y)$ on old code x and new code y, accompanied by a correctness predicate $C(x,y)$ (which may vary slightly across transformations), and a general theorem that T implies C; this theorem is proved once, for each transformation, under user guidance, with effort dependent on the complexity of the transformation. The Solidity compiler has facilities to output Yul abstract syntax trees in JSON format at various stages of the compilation process: these facilities are depicted as the 'export' boxes. We have built a tool, in ACL2, to convert the resulting JSON into abstract syntax trees (ASTs) of our ACL2 formalization of Yul; this is depicted as the 'convert' boxes. Each time the transformation is run, ASTs a and b for the old and new code can be automatically generated; the bottom box can check that $T(a,b)$ holds, and if so it can instantiate the general theorem to obtain a proof of $C(a,b)$ automatically. These last sentences say 'can' because we have not implemented this workflow yet, although we see no obstacle to doing that.

An alternative to verifying the correctness of the replicated transformations is to have them generate proofs of correctness, as done in APT [13, 4, 6] and ATC [3], pushing the verifying compiler approach further. But having pursued that approach in the tools just mentioned, for Yul we wanted to explore the verification of the transformations. In summary, our approach for Yul transformation is a combination of verifying and verified compiler: the former for the transformations in the Solidity compiler, and the latter for the replicated transformations in ACL2.

4.2 Definitions

We have formalized the `ForLoopInitRewriter`, `DeadCodeEliminator`, and `Disambiguator` transformations [19]. The first two are quite simple; the third one is relatively simple, but illustrates a general important point.

The `ForLoopInitRewriter` transformation moves the initialization component of a `for` loop just before the loop and wraps it and the loop in a block, e.g. the loop

```
for { <init> } <test> { <update> } { <body> }
```

is transformed into the block[5]

```
{ <init> for { } <test> { <update> } { <body> } }
```

The scoping rules for `for` loops involve an exception: the scope of variables declared and functions defined in the initialization block extends to the whole loop (test, update, and body). The purpose of performing this transformation is to obviate the need for successive transformations to deal with this scoping exception. This transformation is easily defined in ACL2, by recursion on the abstract syntax.

The `DeadCodeEliminator` transformation removes a simple form of dead code, namely the code in a block that follows a `break`, `continue`, or `leave` statement, e.g. the block

```
{ <live> break <dead> }
```

is transformed into the block[6]

```
{ <live> break }
```

The purpose of this transformation is to reduce code size.

The `Disambiguator` transformation makes all the variable and function names unique across the whole program. For instance, the example code in Section 3.3 is transformed into something like

```
{
  let x
  function f () {
    function h1 () { ... }
    let y1
    {
      let z
    }
  }
  function g () {
    let y2
    function h2 () { ... }
  }
}
```

where the two different variables `y` and functions `h` are renamed apart. There are many ways to rename variables and functions apart, differing in the exact choice of names. To make our ACL2 definition of the transformation simpler, and independent from the choice of names made by the Solidity compiler, we formalized it as a relation instead of a function: the relation holds on old and new code exactly when they are the same except for a consistent renaming of variables and functions such that the new code has globally unique names. In fact, our definition consists of four independent components:

- A binary relation expressing consistent variable renaming.
- A binary relation expressing consistent function renaming.
- A unary relation expressing global uniqueness of variable names.
- A unary relation expressing global uniqueness of function names.

The binary relation for variable renaming consists of a family of recursive functions, such as

```
(define statement-renamevar ((old statementp) (new statementp) (ren renamingp))
  :returns (new-ren renaming-resultp)
```

[5]More precisely, the transformation is recursively applied to `<init>`, `<test>`, `<update>`, and `<body>` as well.

[6]More precisely, the transformation is recursively applied to `<live>` as well.

```
(statement-case
 old
 :block
 (b* (((unless (statement-case new :block)) ...) ; return error
      ((statement-block new) new)
      ((okf &) (block-renamevar old.get new.get ren)))
   (renaming-fix ren))
 :variable-single
 (b* (((unless (statement-case new :variable-single)) ...) ; return error
      ((statement-variable-single new) new)
      ((okf &) (expression-option-renamevar old.init new.init ren)))
   (add-var-to-var-renaming old.name new.name ren))
 ...)) ; handle the other kinds of statements
```

where:

- `old` and `new` are the statements before and after the transformation.
- `ren` is a renaming, i.e. a list of `cons` pairs of identifiers `((x1 . y1) (x2 . y2) ...)` where `x1`, `x2`, etc. are all distinct and where `y1`, `y2`, etc. are all distinct. It is an injective alist with unique keys, invertible into `((y1 . x1) (y2 . x2) ...)`. `x1`, `x2`, etc. are the variables in scope for `old`; `y1`, `y2`, etc. are the variables in scope for `new`. `x1` in `old` is renamed to `y1` in `new`, `x2` in `old` is renamed to `y2` in `new`, etc.; but `x1` and `y1` could be the same, or `x2` and `y2` could be the same, etc.
- If `old` is a block statement, `new` must be a block statement too; otherwise `statement-renamevar` returns an error value, because `new` is not a valid result of transforming `old`.
- The block in `new` must be a valid result of transforming the block in `old`, which is checked by the mutually recursive companion function `block-renamevar`.
- Since a block contributes no new variables outside it, `statement-renamevar` returns `ren`.[7]
- If `old` is a (single) variable declaration, say for `x`, `new` must be one too, say for `y`. The optional initializing expression of `new` must be a valid result of transforming the one of `old`, via the renaming `ren`, which is then extended with the pair `(x . y)`, and returned.
- The code for the other kinds of statements is not shown.

The binary relation for function renaming is defined similarly.

The unary relations for unique variable and function names go through the abstract syntax, keep track of the set of all the names encountered so far, whether visible/accessible or not, and check that every variable declaration or function definition introduces a name not already in the set.

The purpose of `Disambiguator` is to make it easier for subsequent transformations to move code around, without worrying about name conflicts.

Our formalization of `Disambiguator`, unlike our formalizations of the other transformations, does not consist of executable ACL2 code to run the transformation; it consists of executable ACL2 code to check whether the result of the transformation is valid. According to the approach described in Section 4.1, the purpose of formalizing the transformation in ACL2 is to verify that, every time the Solidity compiler runs it, the new code is equivalent to the old code. The old and new code are given as inputs to our formalization of the transformation to check that the action of the Solidity compiler matches our formalization; given a general proof of the correctness of our formalization (see Section 4.4), it is possible to obtain a proof of the correctness of that run of the transformation by the Solidity compiler. This relational approach, which isolates the formal definition and proofs from changeable details of the implementation, may also be applicable to other Yul code transformations.

[7] The fixer `renaming-fix` is a no-op under the guard `(renamingp ren)`, but it makes the return theorem unconditional.

4.3 Restrictions

Some transformations expect the code to satisfy certain restrictions, which must be established by preceding transformations, and must be generally preserved by subsequent transformations. We formalized some of these restrictions in ACL2 as predicates on the abstract syntax.

We formalized the restriction that `for` loops have empty initialization blocks. As mentioned in Section 4.2, this restriction is established by `ForLoopInitRewriter`.

We formalized the restriction that code has no function definitions. Some transformations (not described in this paper) move all the function definitions (after disambiguation) to a new top-level block, so that subsequent transformations can take all the function definitions from the top-level block without worrying about nested function definitions. Our formalized restriction just says that there are no function definitions: it applies to the non-top-level code, which is recursively processed by transformations, stripped of the function definitions at the top level.

4.4 Proofs

We proved that (our formalization of) `DeadCodeEliminator` preserves the two restrictions in Section 4.3. Removing code does not introduce function definitions or code in `for` loop initialization blocks. These proofs are automatic, after enabling the involved functions.

We proved that `DeadCodeEliminator` preserves the static semantics: if the old code is safe, so is the new code, assuming the restriction about the absence of function definitions. The latter is critical: if a function definition follows a `break`, removing the code after the `break` removes that function definition, which the old code may be calling in non-dead code before the `break`. The theorem for statements is

```
(defthm check-safe-statement-of-statement-dead
  (implies (and (statement-nofunp stmt)
                (statement-noloopinitp stmt))
           (b* ((varsmodes (check-safe-statement stmt varset funtab))
                (varsmodes-dead (check-safe-statement (statement-dead stmt)
                                                      varset
                                                      funtab)))
             (implies (not (reserrp varsmodes))
                      (and (not (reserrp varsmodes-dead))
                           (equal (vars+modes->vars varsmodes-dead)
                                  (vars+modes->vars varsmodes))
                           (set::subset (vars+modes->modes varsmodes-dead)
                                        (vars+modes->modes varsmodes)))))))
```

where:

- `statement-dead` removes dead code from a statement, according to the transformation: `stmt` is the old statement, and `(statement-dead stmt)` is the new statement.
- `varset` and `funtab` are the variable and function symbol tables. For this transformation, they are the same for old and new code; more complex transformations may require transforming these tables as well.
- `check-safe-statement`, from the static semantics, checks the safety of a statement; if successful, it returns an updated variable symbol table, and a set of possible termination modes.
- The theorem assumes that: the old statement has no function definitions (critical hypothesis); the old statement has empty `for` loop initialization blocks (which slightly simplifies the proof); the old statement is safe, i.e. `varmodes` is not an error value.

- The theorem concludes that: the new statement is safe, i.e. `varmodes-dead` is not an error value; the updated variable table after the new statement is the same as the one after the old statement; the termination modes of the new statement are a subset of the ones of the old statement (because the static semantics over-approximates them; for example, if a `leave` followed a `break` in the old code, it would be absent in the new code).

The proof is automatic, after enabling the involved functions and also adding an `:expand` hint.[8]

We proved that `DeadCodeEliminator` preserves the dynamic semantics: the new code has the same execution behavior as the old code, assuming the restriction about the absence of function definitions, which is critical for the same reason explained above. The theorem for statements is

```
(defthm exec-statement-of-dead
  (implies (and (statement-nofunp stmt)
                (funenv-nofunp funenv))
           (soutcome-result-okeq
            (exec-statement
             (statement-dead stmt) cstate (funenv-dead funenv) limit)
            (exec-statement stmt cstate funenv limit))))
```

where:

- As in the previous theorem: `stmt` is the old statement; `(statement-dead stmt)` is the new statement; `statement-nofunp` is the critical restriction.
- `funenv-dead` extend the transformation to (the function bodies in) function environments. When a function is called during execution, its body is retrieved from the function environment: to apply induction hypotheses during the proof, the function bodies in the old and new function environments must be related by the transformation, in the same way as the code being executed.
- `funenv-nofunp` extends the restriction of no function definitions to (the function bodies in) function environments. This is also needed to apply induction hypotheses during the proof, because the code of called functions is retrieved from the function environment.
- `soutcome-result-okeq` is an equivalence relation on `soutcome-result` that holds on `a` and `b` exactly when either they are equal statement outcomes or they are both error values. This accommodates slight differences in the details of the error values returned by the dynamic semantics.[9]
- The theorem says that, assuming no function definitions in the old statement and function environment, executing the old statement gives equivalent results to executing the new statement on the same computation state, the transformed function environment, and the same artificial limit.

The proof is not difficult, but involves certain `:expand` and `:use` hints, applied only to certain cases of the induction via computed hints, because they slow down the proof if applied to all the cases. Perhaps the `:use` hints could be avoided in some way, but the `:expand` hints may be necessary to defeat heuristics that prevent the opening of certain recursive function calls.

The formulation of the theorem above does not distinguish between errors due to limit exhaustion and errors due to unsafe operations. In general, a transformation should not turn terminating code into non-terminating code. This can be proved by distinguishing between the two kinds of errors, as done in the theorems for variable renaming, described next.

We proved that the variable renaming component of the `Disambiguator` preserves both static and dynamic semantics. Although this is intuitively obvious, picturing the old and new code merely differing

[8] Without the `:expand` hint, the proof fails, presumably due to heuristics about opening recursive functions.

[9] These slight differences exist because the error values contain some user-oriented information about the error causes, e.g. the constructs that cause the errors. But since this is unnecessary for verification, the error values should probably be simplified, only distinguishing between limit exhaustion and unsafe operations. This should obviate the need for the equivalence relation.

in variable names but otherwise completely isomorphic, it takes a bit of work to formulate and prove.

The theorem for the preservation of the static semantics for statements is

```
(defthm check-safe-statement-when-renamevar
  (b* ((ren1 (statement-renamevar stmt-old stmt-new ren))
       (varmodes-old (check-safe-statement stmt-old (varset-old ren) funtab))
       (varmodes-new (check-safe-statement stmt-new (varset-new ren) funtab)))
    (implies (and (not (reserrp ren1))
                  (not (reserrp varmodes-old)))
             (and (not (reserrp varmodes-new))
                  (equal (vars+modes->vars varmodes-old)
                         (varset-old ren1))
                  (equal (vars+modes->vars varmodes-new)
                         (varset-new ren1))
                  (equal (vars+modes->modes varmodes-old)
                         (vars+modes->modes varmodes-new))))))
```

where:

- The theorem assumes that the new statement `stmt-new` is a valid result of transforming the old statement `stmt-old`, given a renaming `ren`, which results in the possibly extended renaming `ren1`.
- The safety of the old/new statement is checked using the variable symbol table consisting of the keys/values of the renaming `ren`, which are indeed the accessible variables, given how the binary relation `statement-renamevar` is defined (see Section 4.2). The same function symbol table `funtab` is used for both, since functions are not renamed, only variables.[10]
- The theorem assumes that the old statement is safe, and concludes that: the new statement is safe too; the updated variable symbol table after the old/new statement consists of the keys/values of the updated renaming `ren1`; the termination modes of the new statement are the same as the ones of the old statement.

The proof involves some preparatory lemmas, as well as a custom induction scheme that takes into account the recursive structure of both the variable renaming functions like `statement-renamevar` and the static semantic functions like `check-safe-statement`.

The theorem for the preservation of the dynamic semantics for statements is

```
(defthm exec-statement-when-renamevar
  (b* ((ren1 (statement-renamevar stmt-old stmt-new ren)))
    (implies (and (not (reserrp ren1))
                  (cstate-renamevarp cstate-old cstate-new ren)
                  (funenv-renamevarp funenv-old funenv-new))
             (b* ((outcome-old
                   (exec-statement stmt-old cstate-old funenv-old limit))
                  (outcome-new
                   (exec-statement stmt-new cstate-new funenv-new limit)))
               (implies (and (not (reserr-nonlimitp outcome-old))
                             (not (reserr-nonlimitp outcome-new)))
                        (soutcome-result-renamevarp outcome-old
                                                    outcome-new
                                                    ren1))))))
```

where:

- The assumption involving `statement-renamevar` is the same as in the previous theorem.

[10] This is a motivation for decomposing `Disambiguator` into the four independent components described in Section 4.2.

- `cstate-renamevarp` extends variable renaming to computation states, which are built from the variables in the code. The old and new computation states are renamed according to `ren`.
- `funenv-renamevarp` extends variable renaming to function environments, from which the code of called functions is retrieved. This does not depend on `ren`, because the variables in the body of every function in the environment are renamed independently.
- `soutcome-renamevarp` (not directly used in the theorem) extends variable renaming to statement outcomes `a` and `b`: the computation states must be related by `cstate-renamevarp`, and the termination modes must be the same.
- `soutcome-result-renamevarp` extends variable renaming to `soutcome-result`: it holds on `a` and `b` exactly when either they are both statement outcomes satisfying `soutcome-renamevarp` or they are both error values.
- `reserr-nonlimitp` recognizes non-limit error values. Thus, the theorem assumes that the execution of both old and new statement results in either a limit error or a statement outcome.
- The theorem concludes that the two executions yield equivalent results.

The proof involves several preparatory lemmas, a custom induction scheme that takes into account the recursive structure of both the variable renaming functions like `statement-renamevar` and the execution functions like `exec-statement`, and several computed hints to apply different collections of common hints to different cases of the induction.

It may seem strange that the above theorem assumes, instead of concluding, that the execution of the new statement does not yield a non-limit error value: compare the theorem for `DeadCodeEliminator`. However, the preservation of the static semantics by variable renaming described earlier, and the general proof of static soundness in Section 3.5, imply that the execution of the new statement does not yield a non-limit error; thus, it can be assumed in the theorem above, making the proof slightly easier.

The dynamic semantics preservation theorems for `DeadCodeEliminator` require transforming the function environment, but not the computation state or the limit. The dynamic semantics preservation theorems for variable renaming require transforming the computation state and function environment, but not the limit. Theorems for more complex transformations require transforming the limit as well, because the number of execution steps can change.

5 Related Work

We are not aware of any other work on Yul using ACL2.

Other formalizations of Yul exist, written in generic math [14], K [11], Isabelle/HOL [10], Lean [16, 22], and Dafny [8]. As pointed out in [14], none of these are peer-reviewed, except for [14] itself. It does not appear that any of these formalizations include a static semantics separate from the dynamic semantics, and a static soundness proof relating the two. On the other hand, [14] includes both a small-step and a big-step dynamic semantics, with a proof of equivalence. An advantage of [14] compared to the other formalizations, including ours, is that it is written in a generic mathematical notation that is more widely accessible than the language and libraries of ACL2 and similar tools; a disadvantage is that it is not machine-checked, unlike the ones developed with ACL2 and similar tools.

The `README` in the K formalization of Yul [11] says that its purpose is to perform translation validation of the Solidity compiler, which is exactly the same goal as ours; that `README` also indicates scripts to run tests. It would be interesting to compare their work with ours, but the lack of published papers and detailed documentation demands an examination of their code and prerequisite knowledge of K. The stated purpose of [14] is to provide a widely accessible precise formalization of Yul. The purpose of [10, 16, 8]

appears to be mainly the formal verification of Yul code, but not specifically Yul transformations.

6 Future Work

Although our formalization of the Yul core is essentially complete, it hard-wires some aspects of the EVM dialect, which should better be kept more separate via a more explicit parameterization of the core formalization over the dialect. This may take some effort because, despite the multi-dialect aspiration, currently the Yul core and the EVM dialect are not crisply delineated, and manifest very much like one integrated language; for instance, different dialects are supposed to have different type systems, but the Yul syntax does not provide a way to specify types.[11]

More importantly than having a cleaner separation between core and dialects, the EVM dialect should be formalized completely, since it is currently the only Yul dialect in practical use. This is a laborious task because it involves modeling a large portion of the functionality of the EVM [26], although it does not require a full formalization of the EVM itself.

Our work on transformations has barely scratched the surface. The Solidity compiler includes tens of transformations, some of which are rather complex and involve EVM-dialect-specific features. Formalizing and verifying all of them is a substantial task, but it can be approached piecewise: each transformation can be formalized and verified mostly on its own; dependencies among transformations can be handled by formalizing the kinds of restrictions on Yul code exemplified in Section 4.3.

To verify the transformations performed by the Solidity compiler, the theorem generator depicted in Figure 2 must be developed. Once all transformations have been formalized and verified, the generated proofs can be composed into a proof that the entire sequence of Yul transformations was correctly applied.

7 Conclusion

Our formalization of Yul, like formalizations developed by others, does not contain any particularly innovative ideas, partly because Yul is a relatively simple language. However, a precise, machine-checked formalization of Yul is clearly valuable. Furthermore, it is possible that extending the formalization to the full EVM dialect may uncover more interesting formalization issues.

The work on verifying Yul transformations is more original, though there is some similar work (see Section 5). An interesting finding of our work was that verifying the correctness of even fairly simple transformations, such as those described in Section 4, whose correctness intuitively appears obvious, required more work than expected. The proofs are not difficult, but a bit laborious.

Acknowledgements

We thank the Ethereum Foundation for supporting this work.

[11] The Yul documentation includes an older grammar with syntax for type identifiers, and a newer grammar without such syntax. The Yul team from the Ethereum Foundation told us that the latter supersedes the former.

References

[1] Alessandro Coglio (2004): *Simple Verification Technique for Complex Java Bytecode Subroutines*. *Concurrency and Computation: Practice and Experience* 16(7), pp. 647–670, doi:10.1002/cpe.798.

[2] Alessandro Coglio (2018): *A Formalization of the ABNF Notation and a Verified Parser of ABNF Grammars*. In: *Proc. 10th Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*, *Lecture Notes in Computer Science (LNCS)* 11294, pp. 177–195, doi:10.1007/978-3-030-03592-1_10.

[3] Alessandro Coglio (2022): *A Proof-Generating C Code Generator for ACL2 Based on a Shallow Embedding of C in ACL2*. In: *Proc. 17th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2022)*, *Electronic Proceedings in Theoretical Computer Science (EPTCS)* 359, pp. 185–201, doi:10.4204/EPTCS.359.15.

[4] Alessandro Coglio, Matt Kaufmann & Eric Smith (2017): *A Versatile, Sound Tool for Simplifying Definitions*. In: *Proc. 14th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2017)*, *Electronic Proceedings in Theoretical Computer Science (EPTCS)* 249, pp. 61–77, doi:10.4204/EPTCS.249.5.

[5] Alessandro Coglio, Eric McCarthy & Eric Smith (2022): *Formal Verification of Zero-Knowledge Circuits*. In: *Proc. 18th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2022)*, *Electronic Proceedings in Theoretical Computer Science (EPTCS)* 393, pp. 94–112, doi:10.4204/EPTCS.393.9.

[6] Alessandro Coglio & Stephen Westfold (2020): *Isomorphic Data Type Transformations*. In: *Proc. 16th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2020)*, *Electronic Proceedings in Theoretical Computer Science (EPTCS)* 327, pp. 125–141, doi:10.4204/EPTCS.327.12.

[7] D. Crocker & P. Overell (2008): *Augmented BNF for Syntax Specifications: ABNF*. Request for Comments (RFC) 5234.

[8] *Yul in Dafny*. Available at `https://github.com/franck44/yul-dafny`.

[9] *The Ethereum Blockchain*. Available at `https://ethereum.org`.

[10] *Yul in Isabelle/HOL*. Available at `https://github.com/mmalvarez/Yul-Isabelle`.

[11] *Yul in K*. Available at `https://github.com/ethereum/Yul-K`.

[12] Matt Kaufmann & J Strother Moore: *The ACL2 Theorem Prover*. Available at `http://acl2.org`.

[13] Kestrel Institute: *APT (Automated Program Transformations)*. `https://www.kestrel.edu/research/apt`.

[14] Vasileios Koutavas, Yu-Yang Lin & Nikos Tzevelekos (2024): *An Operational Semantics for Yul*. In Alexandre Madeira & Alexander Knapp, editors: *Proc. 22nd International Conference on Software Engineering and Formal Methods (SEFM)*, *Lecture Notes in Computer Science (LNCS)* 15280, pp. 328–346, doi:10.1007/978-3-031-77382-2_19.

[15] P. Kyzivat (2014): *Case-Sensitive String Support in ABNF*. Request for Comments (RFC) 7405.

[16] *Yul in Lean*. Available at `https://github.com/NethermindEth/Yul-Specification`.

[17] *The Solidity Compiler*. Available at `https://github.com/ethereum/solidity`.

[18] *The Solidity Documentation*. Available at `https://docs.soliditylang.org`.

[19] *The Solidity Documentation: The Optimizer*. Available at `https://docs.soliditylang.org/en/latest/internals/optimizer.html`.

[20] *The Solidity Documentation: Yul*. Available at `https://docs.soliditylang.org/en/latest/yul.html`.

[21] *The Solidity Language*. Available at `https://soliditylang.org`.

[22] Julian Sutherland (2022): *Securing Warp: A formal specification of the Yul IR*. Available at `https://medium.com/nethermind-eth/securing-warp-a-formal-specification-of-the-yul-ir-85bb3bf51c62`. Medium post.

[23] Sol Swords & Jared Davis (2015): *Fix Your Types*. In: *Proc. 13th International Workshop on the ACL2 Theorem Prover and Its Applications*, pp. 3–16, doi:10.4204/EPTCS.192.2.

[24] The ACL2 Community: *The ACL2 Theorem Prover and Community Books: Documentation*. Available at `http:/acl2.org/manual`.

[25] The ACL2 Community: *The ACL2 Theorem Prover and Community Books: Source Code*. Available at `http://github.com/acl2/acl2`.

[26] Gavin Wood: *Ethereum: A Secure Decentralized Generalised Transaction Ledger*. `https://ethereum.github.io/yellowpaper/paper.pdf`.

A Formalization of the Correctness of the Floodsub Protocol

Ankit Kumar Panagiotis Manolios
Northeastern University
Boston, USA
{kumar.anki,p.manolios}@northeastern.edu

Floodsub is a simple, robust and popular peer-to-peer publish/subscribe (pubsub) protocol, where nodes can arbitrarily leave or join the network, subscribe to or unsubscribe from topics and forward newly received messages to all of their neighbors, except the sender or the originating peer. To show the correctness of Floodsub, we propose its specification: Broadcastsub, in which implementation details like network connections and neighbor subscriptions are elided. To show that Floodsub does really implement Broadcastsub, one would have to show that the two systems have related infinite computations. We prove this by reasoning locally about states and their successors using Well-Founded Simulation (WFS). In this paper, we focus on the mechanization of a proof which shows that Floodsub is a simulation refinement of Broadcastsub using WFS. To the best of our knowledge, ours is the first mechanized refinement-based verification of a real world pubsub protocol.

1 Introduction

Peer-to-Peer (P2P) systems are decentralized distributed systems, which constitute overlay networks built over physical networks, such as the Internet [2]. These systems are characterized by self-organization, being able to handle highly dynamic network configurations, with nodes being able to join or leave the overlay network, which allows for scalability in the size of the networks. Publish/Subscribe (*pubsub*) systems are P2P systems that allow (1) consumers of information (subscriber nodes) to query the system, and (2) producers of information (publisher nodes) to publish information to the system. Publishers are able to send messages to multiple recipients without them having to know who the subscribers are. This is achieved by associating subscriptions and messages with *topics*. For example, in a chat room application, each room is a pubsub topic and clients post chat messages to rooms, which are received by all other clients (subscribers) in the room. The Scribe system [5] was a first attempt at providing topic-based pubsub functionality over the Pastry P2P network [33].

A most basic implementation of pubsub systems is *Floodsub* [36, 1]. In Floodsub, nodes are free to leave or join the network. Every node has information about its neighboring nodes and their subscriptions. Whenever a node joins, subscribes to or unsubscribes from a topic, it updates its neighboring nodes. Messages are forwarded to all neighbors that subscribe to the topic of the message, except the source (originating node) of the message, or the node that forwarded this message. Our implementation of Floodsub is based on its specification [36]. However, we did not do any conformance testing or cross validation against existing implementations. And we do not model Ambient Peer Discovery, which is a way for nodes to learn about their neighbors, and is described in the specification as being external to the protocol. Given our model of Floodsub protocol, how can we verify that it actually is an implementation of a pubsub system? We need a specification for a pubsub system and some notion of correctness.

We propose *Broadcastsub* [1] as the specification for a P2P pubsub system. Broadcastsub nodes can freely leave and join the network. Nodes maintain a list of topics they subscribe to. However, there is no notion of neighboring nodes. Messages are broadcasted "magically" to all the subscribers in a single

Gamboa and Manolios (Eds):
ACL2 Workshop 2025
EPTCS 423, 2025, pp. 84–103, doi:10.4204/EPTCS.423.9

transition. Notice that Broadcastsub is the simplest P2P pubsub system, where implementation details like subscription updates, neighboring nodes and their subscriptions are abstracted. In this paper, we focus on the mechanization of the proof that Floodsub is a simulation refinement [31] of Broadcastsub using Well-Founded Simulation (WFS) [23] in ACL2 Sedan (ACL2s) [12, 6].

To show that Floodnet implements Broadcastnet, we will prove that Floodnet is a simulation refinement of Broadcastnet. Why are we proving a simulation refinement? Because we are comparing two P2P systems at different levels of abstraction. In a Broadcastsub network, a message broadcast propagates to all subscribers (of the message) instantly. However, in a Floodsub network, a message may require several hops from one node to another, until it reaches all of the subscribers. It is often the case that a lower-level implementation takes several steps to match a step of its higher level specification. Proving a WFS guarantees that Floodsub states and related Broadcastsub states have related computations. This notion of correctness implies that the two systems satisfy the same $ACTL^* \setminus X$ [4] properties. WFS proofs are structural and local, requiring proofs about states and their successors, instead of infinite paths, thereby allowing proofs to be amenable to formal verification. This work is a piece of a larger puzzle that allows us to reason about more complex P2P systems using compositional refinement [25, 24], which we want to extend all the way down to Gossipsub [35]. Since our models are public, protocol engineers will be able to easily define/extend their own P2P systems and attempt to show that their model is a refinement of one of our existing ones. Our proof of refinement spells out exactly how the proof breaks, if these conditions are not satisfied. Hence, our contribution can also be used to tag P2P systems with the kinds of network attacks they are prone to, corresponding to the refinement conditions that were not satisfied.

We make the following contributions: (1) Formal, executable, open and public models of Floodsub and Broadcastsub protocols expressed as transition systems, and (2) a mechanized proof in ACL2s showing that Floodsub is a simulation refinement of Broadcastsub. We discuss related work in Section 5; and ours is the first mechanized refinement-based verification of a real world pubsub protocol. While refinement is a standard formal method, it has never been previously applied to P2P pubsub protocols like FloodSub. Our models and proofs are publicly available in our repository [21]. Overall, our code consists of 476 theorems proved, and 10277 lines of lisp code.

Paper Outline. Section 2 describes Floodnet and Broadcastnet models for Floodsub and Broadcastsub, respectively. Section 3 describes the refinement theorem. Section 4 is a discussion about the theorem proving process and effort that went into this proof. Section 5 discusses related work. Section 6 concludes.

2 Model Descriptions

We model Floodsub and its specification Broadcastsub using transition systems consisting of states and transition relations (boolean functions on 2 states) that depend on transition functions. We call our models Floodnet and Broadcastnet respectively. In this section we will explain each of our transition system models in a top-down fashion. The state models are self explanatory. The interesting parts of the following code listings are the transition relations, where we place conditions, not only as sanity checks or for cases, but also as guards to disallow illegal behaviour, like requiring that a peer leaving a network is already in the network. We will discuss such conditions in detail.

2.1 Broadcastnet

The state of a Broadcastnet is stored in a map from peers to their corresponding peer-states. We represent peers by natural numbers. A Broadcastnet peer state is a record consisting of (i) `pubs` : the set of topics in which a peer publishes, (ii) `subs` : the set of topics to which a peer subscribes to, and (iii) `seen` : the set of messages a peer has already processed. We use sorted ACL2 lists containing unique elements to represent sets. Hence, set equality is reduced to list equality. Messages are records consisting of (i) `pld` : a message payload of type string (ii) `tp` : the topic in which this message was published, and (iii) `or` : the originating peer for this message.

```
(defdata s-bn (map peer ps-bn))

(defdata-alias peer nat)

(defdata ps-bn (record
                (pubs . lot)
                (subs . lot)
                (seen . lom)))

(defdata lot (listof topic))

(defdata-alias topic var)

(defdata lom (listof mssg))

(defdata mssg (record
               (pld . string)
               (tp . topic)
               (or . peer)))
```

We will now define the transition relation for Broadcastnet. The relation `rel-step-bn` relates 2 Broadcastnet states `s` and `u` iff `s` transitions to `u`. `rel-step-bn` is an `OR` of all the possible ways `s` can transition to `u`. `rel-skip-bn` represents a transition where `s` chooses to skip, hence `u` is `s`.

```
(definec rel-step-bn (s u :s-bn) :bool
  (v (rel-skip-bn s u)
     (rel-broadcast-bn s u)
     (rel-broadcast-partial-bn s u)
     (rel-subscribe-bn s u)
     (rel-unsubscribe-bn s u)
     (rel-leave-bn s u)
     (rel-join-bn s u)))

(definecd rel-skip-bn (s u :s-bn) :bool
  (== u s))
```

`rel-broadcast-bn` defines a relation between `s` and `u`, where `u` represents the state resulting from broadcasting a message in `s`. The broadcast is modeled as an atomic operation in which all subscribers receive the message simultaneously. The function `(br-mssg-witness s u)` is a witness finding function that calculates the message that was broadcast, if one exists. Since `seen` is a set of messages implemented as an ordered list with unique elements, `br-mssg-witness` utilizes this ordering of unique messages to find the broadcasted message.

The boolean function `broadcast-bn-pre` is a conjunction of the following preconditions: (i) the broadcast message is new, *i.e.*, it is not already found in the `seen` set of any of the peers in `s`, (ii) the originating peer of the message exists in `s`, and (iii) the topic of the broadcast message is one in which the originating peer publishes messages. `broadcast-bn-pre` also appears as an input contract (`:ic`) in the definition of the `broadcast` transition function. `(== u (broadcast (br-mssg-witness s u) s))` in the definition of `rel-broadcast-bn` ensures that the message found by `br-mssg-witness` was the sole message broadcast in `s`. We use the `insert-unique` function within `broadcast-help` to add new messages while preserving order and uniqueness of the `seen` set.

In the following code snippets, we use some ACL2s syntax described as follows. `^`, `v` and `!` are macros for `and`, `or` and `not` respectively. In a `property` form, the form following the keyword `:h` is the hypothesis, while the form following the keyword `:b` is the body. `(nin a x)` stands for `(not (in a x))`. `:match` is a powerful ACL2s pattern matching capability which supports predicates, including recognizers automatically generated by `defdata`, disjunctive patterns and patterns containing arbitrary code [30]. For more expressive pattern matching, `!` is used for literal match while `&` is used as a wildcard. In a function definition form, `:ic` and `:oc` abbreviate `:input-contract` and `:output-contract` respectively.

```
(definecd rel-broadcast-bn (s u :s-bn) :bool
  (^ (br-mssg-witness s u)
     (broadcast-bn-pre (br-mssg-witness s u) s)
     (== u (broadcast (br-mssg-witness s u) s))))

(definec broadcast-bn-pre (m :mssg s :s-bn) :bool
  (b* ((origin (mget :or m))
       (origin-st (mget origin s)))
    (^ (new-bn-mssgp m s)
       origin-st
       (in (mget :tp m)
           (mget :pubs origin-st)))))

(definec br-mssg-witness (s u :s-bn) :maybe-mssg
  (cond
   ((v (endp s) (endp u)) nil)
   ((== (car s) (car u)) (br-mssg-witness (cdr s) (cdr u)))
   (t (car (set-difference-equal (mget :seen (cdar u))
                                 (mget :seen (cdar s)))))))

(defdata maybe-mssg (v nil mssg))

(definecd new-bn-mssgp (m :mssg s :s-bn) :bool
  (v (endp s)
     (^ (nin m (mget :seen (cdar s)))
        (new-bn-mssgp m (cdr s)))))

(definecd broadcast (m :mssg s :s-bn) :s-bn
  :ic (broadcast-bn-pre m s)
  (broadcast-help m s))

(definecd broadcast-help (m :mssg st :s-bn) :s-bn
  (match st
```

```
   (() nil)
   (((p . pst) . rst)
    (cons `(,p . ,(if (v (in (mget :tp m) (mget :subs pst))
                         (== p (mget :or m)))
                      (mset :seen
                            (insert-unique m (mget :seen pst))
                            pst)
                    pst))
          (broadcast-help m rst)))))

(definec insert-unique (a :all x :tl) :tl
  (match x
    (() (list a))
    ((!a . &) x)
    ((e . es) (if (<< a e) (cons a x) (cons e (insert-unique a es))))))
```

We will explain `rel-broadcast-partial-bn`, and its necessity when discussing the proof of correctness later in the paper. `rel-subscribe-bn` and `rel-unsubscribe-bn` relate states `s` and `u` where `u` represents the state obtained after a peer in `s` subscribes to or unsubscribes from a set of topics, respectively. `bn-topics-witness` calculates the peer and the set of topics it subcribes to or unsubscribes from, if there exists such peer. Notice that we reuse `bn-topics-witness` in the definition of `rel-unsubscribe-bn`, with the arguments reversed, so as to find the topics that are subscribed to in `s`, but not in `u`. The calculated set of topics are unioned with or removed from the existing set of peer topic subscriptions of the calculated peer, based on whether it is subscribing or unsubscribing. The definition of `unsubscribe-bn` is analogous to that of `subscribe-bn` and is hence omitted.

```
(definecd rel-subscribe-bn (s u :s-bn) :bool
  (^ (bn-topics-witness s u)
     (mget (car (bn-topics-witness s u)) s)
     (== u (subscribe-bn (car (bn-topics-witness s u))
                         (cdr (bn-topics-witness s u))
                         s))))

(definecd rel-unsubscribe-bn (s u :s-bn) :bool
  (^ (bn-topics-witness u s)
     (mget (car (bn-topics-witness u s)) s)
     (== u (unsubscribe-bn (car (bn-topics-witness u s))
                           (cdr (bn-topics-witness u s))
                           s))))

(definec bn-topics-witness (s u :s-bn) :maybe-ptops
  (cond
   ((v (endp s) (endp u)) nil)
   ((== (car s) (car u)) (bn-topics-witness (cdr s) (cdr u)))
   ((^ (== (caar s) (caar u))
       (set-difference-equal (mget :subs (cdar u))
                             (mget :subs (cdar s))))
    (cons (caar s)
          (set-difference-equal (mget :subs (cdar u))
                                (mget :subs (cdar s)))))
   (t nil)))
```

```
(defdata maybe-ptops (v nil (cons peer lot)))

(definecd subscribe-bn (p :peer topics :lot s :s-bn) :s-bn
  :ic (mget p s)
  (let ((pst (mget p s)))
    (mset p (mset :subs (union-equal (mget :subs pst) topics) pst) s)))
```

`rel-join-bn` and `rel-leave-bn` relate states `s` and `u` where `u` is obtained after a peer joins `s` or leaves `s`, respectively. `bn-join-witness` calculates the peer and its peer-state, if there exists a peer that joins `s`. Its definition depends on the keys of our Broadcastnet state being in order, which is guaranteed by ACL2s `maps`. A peer joins a Broadcastnet state when a new default Broadcastnet peer state is set for the corresponding peer. A peer leaves a Broadcastnet state when the entry corresponding to the leaving peer is removed from the state.

```
(definecd rel-join-bn (s u :s-bn) :bool
  (^ (bn-join-witness s u)
     (b* ((p (car (bn-join-witness s u)))
          (pst (cdr (bn-join-witness s u))))
       (^ (! (mget p s))
          (== u (join-bn p (mget :pubs pst) (mget :subs pst) s))))))

(definecd rel-leave-bn (s u :s-bn) :bool
  (^ (bn-join-witness u s)
     (mget (car (bn-join-witness u s)) s)
     (== u (leave-bn (car (bn-join-witness u s)) s))))

(definec bn-join-witness (s u :s-bn) :maybe-ppsbn
  (match (list s u)
    ((() ((q . qst) . &)) `(,q . ,qst))
    ((((p . pst) . rs1) ((q . qst) . rs2))
     (cond
      ((== `(,p . ,pst) `(,q . ,qst)) (bn-join-witness rs1 rs2))
      ((!= q p) `(,q . ,qst))  ;; Joining peer found
      (t nil)))
    (& nil)))

(defdata maybe-ppsbn (v nil (cons peer ps-bn)))

(definecd join-bn (p :peer pubs subs :lot s :s-bn) :s-bn
  :ic (! (mget p s)) ;; Join only if peer does not already exist in state
  (mset p (ps-bn pubs subs '()) s))

(definecd leave-bn (p :peer s :s-bn) :s-bn
  :ic (mget p s) ;; Leave only if peer already exists in state
  (match s
    (((!p . &) . rst) rst)
    ((r . rst) (cons r (leave-bn p rst)))))
```

2.2 Floodnet

A Floodnet peer-state is a record consisting of sets `pubs`, `subs` and `seen` which we described previously in context of Broadcastnet peer-states. It also consists of `pending`, which is a set of messages that have not yet been processed, and `nsubs`, a map from topics to list of peers. `nsubs` stores topic subscriptions for neighboring peers.

```
(defdata s-fn (map peer ps-fn))

(defdata ps-fn
  (record (pubs . lot)
          (subs . lot)
          (nsubs . topic-lop-map)
          (pending . lom)
          (seen . lom)))
```

When a message has not been forwarded to neighboring subscribers (processed) it remains in the `pending` set. Once it is processed, it is added to the `seen` set. In our Floodnet model, `pending` and `seen` are sets of messages, instead of queues. This simplifies the model and allows us to not worry about the order in which messages are received. Related states have equal sets of `seen` messages.

We define the transition relation `rel-step-fn` which relates two Floodnet states `s` and `u` iff `s` transitions to `u`. It encodes all the possible ways `s` can transition to `u`. `rel-skip-fn` represents a transition where `s` chooses to skip, hence `u` is `s`.

```
(definec rel-step-fn (s u :s-fn) :bool
  (v (rel-skip-fn s u)
     (rel-produce-fn s u)
     (rel-forward-fn s u)
     (rel-subscribe-fn s u)
     (rel-unsubscribe-fn s u)
     (rel-leave-fn s u)
     (rel-join-fn s u)))

(definecd rel-skip-fn (s u :s-fn) :bool
  (== u s))
```

`rel-produce-fn` relates `s` and `u` where `u` represents the state obtained after a new message has been produced in `s`. The newly produced message is one of the pending messages in `u`. The boolean function `produce-fn-pre` is a conjunction of the following preconditions: (i) the produced message is new *i.e.*, it is not already found in the `seen` or `pending` sets of any of the peers in `s`, (ii) the originating peer of the message exists in `s`, and (iii) the topic of the produced message is one in which the originating peer publishes messages. The new message is added to the set of pending messages of the originating peer.

```
(definecd rel-produce-fn (s u :s-fn) :bool
  (rel-produce-help-fn s u (fn-pending-mssgs u)))

(definec fn-pending-mssgs (s :s-fn) :lom
  (match s
    (() '())
    (((& . pst) . rst) (union-set (mget :pending pst) (fn-pending-mssgs rst)))))

(definec rel-produce-help-fn (s u :s-fn ms :lom) :bool
  (match ms
```

```
    (() nil)
    ((m . rst) (v (^ (produce-fn-pre m s)
                     (== u (produce-fn m s)))
                  (rel-produce-help-fn s u rst)))))

(definec produce-fn-pre (m :mssg s :s-fn) :bool
  (b* ((origin (mget :or m))
       (origin-st (mget origin s)))
    (^ (new-fn-mssgp m s)
       origin-st
       (in (mget :tp m) (mget :pubs origin-st)))))

(definecd new-fn-mssgp (m :mssg s :s-fn) :bool
  (v (endp s)
     (^ (nin m (mget :seen (cdar s)))
        (nin m (mget :pending (cdar s)))
        (new-fn-mssgp m (cdr s)))))

(definecd produce-fn (m :mssg s :s-fn) :s-fn
  :ic (produce-fn-pre m s)
  (mset (mget :or m)
        (add-pending-psfn m (mget (mget :or m) s)) s))

(definecd add-pending-psfn (m :mssg pst :ps-fn) :ps-fn
  (if (v (in m (mget :pending pst))
         (in m (mget :seen pst)))
      pst
    (mset :pending (cons m (mget :pending pst)) pst)))
```

`rel-forward-fn` relates states `s` and `u` where `u` represents the state obtained after a peer in `s` forwards a pending message. Notice that any of the pending messages in `s` are eligible to be forwarded. Notice also that there can be several peers with a given message pending, and the Floodnet can take several possible transitions to states related to the current state by `rel-forward-fn`. To model this in a constructive and deterministic way, we introduce `find-forwarder` as a skolem function which returns the first peer in the state where a given message is pending. It produces a concrete peer p in the call to `forward-fn`. Its output contract (`:oc`) specifies that the message forwarding peer it returns (i) is a peer in `s`, (ii) possesses the given message in its `pending` set, and (iii) that message is not new in `s`.

The `forward-fn` transition function simultaneously updated the state of the peer that forwards the message, using `update-forwarder-fn` and updates the `pending` sets of the neighboring subscribers by inserting the forwarded message using `forward-help-fn`. Note that messages are forwarded to all the peers subscribing to the topic of the message in the `:nsubs` map. If the forwarding peer records its own subscriptions in `:nsubs`, it can lead to `rel-forward-fn` being infinitely enabled. We will ensure that a peer does not include itself in this map, by considering *good* Floodnet states later in the paper.

```
(definecd rel-forward-fn (s u :s-fn) :bool
    (rel-forward-help-fn s u (fn-pending-mssgs s)))

(definec rel-forward-help-fn (s u :s-fn ms :lom) :bool
  (match ms
    (() nil)
    ((m . rst)
```

```
    (v (^ (in m (fn-pending-mssgs s))
          (== u (forward-fn (find-forwarder s m) m s)))
       (rel-forward-help-fn s u rst)))))

(definec find-forwarder (s :s-fn m :mssg) :peer
   :ic (in m (fn-pending-mssgs s))
   :oc (^ (mget (find-forwarder s m) s)
          (in m (mget :pending (mget (find-forwarder s m) s)))
          (! (new-fn-mssgp m s)))
   (match s
     (((p . &)) p)
     (((p . pst) . rst)
      (if (in m (mget :pending pst)) p (find-forwarder rst m)))))

(definecd forward-fn (p :peer m :mssg s :s-fn) :s-fn
  :ic (^ (mget p s)
         (in m (mget :pending (mget p s))))
  (b* ((tp (mssg-tp m))
       (pst (mget p s))
       (nsubs (mget :nsubs pst))
       (fwdnbrs (mget tp nsubs)))
    (forward-help-fn (update-forwarder-fn p m s) fwdnbrs m)))

(definec update-forwarder-fn (p :peer m :mssg s :s-fn) :s-fn
  (match s
    (() '())
    (((!p . pst) . rst) (cons `(,p . ,(forwarder-new-pst pst m)) rst))
    ((r . rst) (cons r (update-forwarder-fn p m rst)))))

(definecd forwarder-new-pst (pst :ps-fn m :mssg) :ps-fn
  (mset :seen
        (insert-unique m (mget :seen pst))
        (mset :pending
              (remove-equal m (mget :pending pst))
              pst)))

(definecd forward-help-fn (s :s-fn nbrs :lop m :mssg) :s-fn
  (match s
    (() '())
    (((q . qst) . rst)
     (cons (if (in q nbrs)
               `(,q . ,(add-pending-psfn m qst))
             `(,q . ,qst))
           (forward-help-fn rst nbrs m)))))
```

`rel-subscribe-fn` and `rel-unsubscribe-fn` relate states `s` and `u` where `u` represents the state obtained after a peer in `s` subscribes to or unsubscribes from a set of topics, respectively. They are very similar to their Broadcastnet counterparts and hence we omit their definitions.

`rel-join-fn` and `rel-leave-fn` relate states `s` and `u` where `u` represents the state obtained after a peer joins `s` or leaves `s`, respectively. `fn-join-witness` calculates the peer and its peer-state, if there exists a peer that joins `s`, and is analogous to `bn-join-witness`. `rel-join-fn` requires that the joining

peer (i) is not already in `s`, and (ii) does not exist in its own `:nsubs` map. The second condition is necessary to prevent peers from endlessly forwarding messages to themselves. `join-fn` depends on `new-joinee-st-fn` which returns the state for a newly joined peer, and on `set-subs-sfn` which updates the `:nsubs` map for each of the neighboring peers of the joining node. For the sake of brevity, we omit the definitions of these helper functions. The `rel-leave-fn` relation, similar to `rel-leave-bn` requires that there is a leaving peer, as calculated by `(fn-join-witness u s)` and that it already exist in the state. Notice that there is another requirement, that a leaving peer has no pending messages. This condition allows for graceful exit of leaving peers, guaranteeing that no pending messages are lost along with them.

```
(definecd rel-join-fn (s u :s-fn) :bool
  (^ (fn-join-witness s u)
     (b* ((p (car (fn-join-witness s u)))
          (pst (cdr (fn-join-witness s u)))
          (nbrs (topic-lop-map->lop (mget :nsubs pst))))
       (^ (! (mget p s))
          (nin p nbrs)
          (== u (join-fn p (mget :pubs pst) (mget :subs pst) nbrs s))))))

(definecd join-fn (p :peer pubs subs :lot nbrs :lop s :s-fn) :s-fn
  :ic (^ (! (mget p s))
         (nin p nbrs))
  (set-subs-sfn nbrs
                subs
                p
                (mset p (new-joinee-st-fn pubs subs nbrs s) s)))

(definecd rel-leave-fn (s u :s-fn) :bool
  (^ (fn-join-witness u s)
     (mget (car (fn-join-witness u s)) s)
     (endp (mget :pending (mget (car (fn-join-witness u s)) s)))
     (== u (leave-fn (car (fn-join-witness u s)) s))))

(definecd leave-fn (p :peer s :s-fn) :s-fn
  :ic (mget p s)
  (match s
    (() '())
    (((!p . &) . rst) rst)
    ((r . rst) (cons r (leave-fn p rst)))))
```

3 Correctness and the Refinement Theorem

We consider simulation refinement [23] as the notion of correctness for Floodnet and show that Floodnet is a simulation refinement of Broadcastnet. The key idea of a simulation refinement is to show that every behavior of the concrete system (Floodnet) is allowed by the abstract system (Broadcastnet). If we prove a WFS refinement then we know that for any infinite computation tree starting from some Floodnet state, we can find a related computation tree in Broadcastnet after applying the refinement map. Another consequence is that we preserve any branching time properties, excluding next time, for example, all properties in $ACTL^* \setminus X$ [4].

The refinement map needs to map Floodnet states to "related" Broadcastnet states. Why do we require a refinement map? Because states in different levels of abstractions may represent data differently, or some implementation details from the lower abstraction may simply be missing in the higher level specification. For example, `:nsubs` and `:pending` appear only in Floodnet peer states, not in Broadcastnet peer states. Using a refinement map is like putting on glasses that let us "see" lower-level concrete states as their corresponding abstract specification states. The refinement map that we use is `f2b` shown below. It maps Floodnet states to Broadcastnet states where pending messages have not yet been broadcasted. This is called as the commitment approach to refinement, since we are mapping to states consisting of only those messages that have been fully propagated in Floodnet and are thus considered committed.

```
(definec f2b (s :s-fn) :s-bn
  (f2b-help s (fn-pending-mssgs s)))

(definec f2b-help (s :s-fn ms :lom) :s-bn
  (if (endp s)
      '()
    (cons `(,(caar s) . ,(f2b-st (cdar s) ms))
          (f2b-help (cdr s) ms))))

(definecd f2b-st (ps :ps-fn ms :lom) :ps-bn
  (ps-bn (mget :pubs ps)
         (mget :subs ps)
         (set-difference-equal (mget :seen ps) ms)))
```

To gain a better understanding of our refinement map, we examine example traces of Floodnet and Broadcastnet in Figure 1. On the left side, we have a trace of a Floodnet, consisting of 3 green colored nodes, numbered 1, 2 and 3. The node numbered 3 is connected to nodes 1 and 2. We show pending messages on the top left of a node, and seen messages on the bottom right. So, in the second Floodnet state shown, node 1 has a pending message `m`, after a `produce-fn` transition. On the right side, we have Broadcastnet states such that for each Floodnet state on the left, we have its refinement map on the right and for each transition on the left, we show a corresponding matching transition on the right.

The transitions on the Floodnet side are as follows : (i) Node 1 produces message `m`; (ii) Node 1 forwards its pending message `m` to its connected neighboring peer 3; (iii) Node 1 leaves the network (iv) Node 2 unsubscribes from `(mssg-tp m)`, which is the message topic (iv) Node 3 unsubscribes from `(mssg-tp m)`, and finally (v) Node 3 forwards `m` to node 2. Notice that `f2b` is a clear refinement map where events like joining and leaving are not masked. Hence, in the corresponding Broadcastnet states, leave and unsubscribe transitions are matched with leave and unsubscribe transitions. However, when the message `m` can no longer be forwarded, and is no longer pending, it needs to be matched by a `broadcast`. But notice that on the Broadcastnet side (a) the originating peer (Node 1) is no longer present, which is required for a `broadcast`, due to `broadcast-bn-pre`, and (b) there are no subscribers of `m` left in the network! This issue arises from the fact that broadcasting a message in Floodnet is a highly fragmented operation, taking place over several message hops during which peers are free to leave, join, subscribe or unsubscribe. With so many moving parts in the network, it becomes impossible to specify which nodes will receive the broadcasted message in the Broadcastnet under the refinement map at the time the message is produced. To solve this problem, we generalize the Broadcastnet specification by adding another transition relation: `rel-broadcast-partial-bn` which allows us to relate 2 states `s` and `u` where `u` represents the state obtained after broadcasting a message in `s`, but only partially. This relation is

defined using the `broadcast-partial` transition function, which given a message and a list of peers, sends the message to those peers.

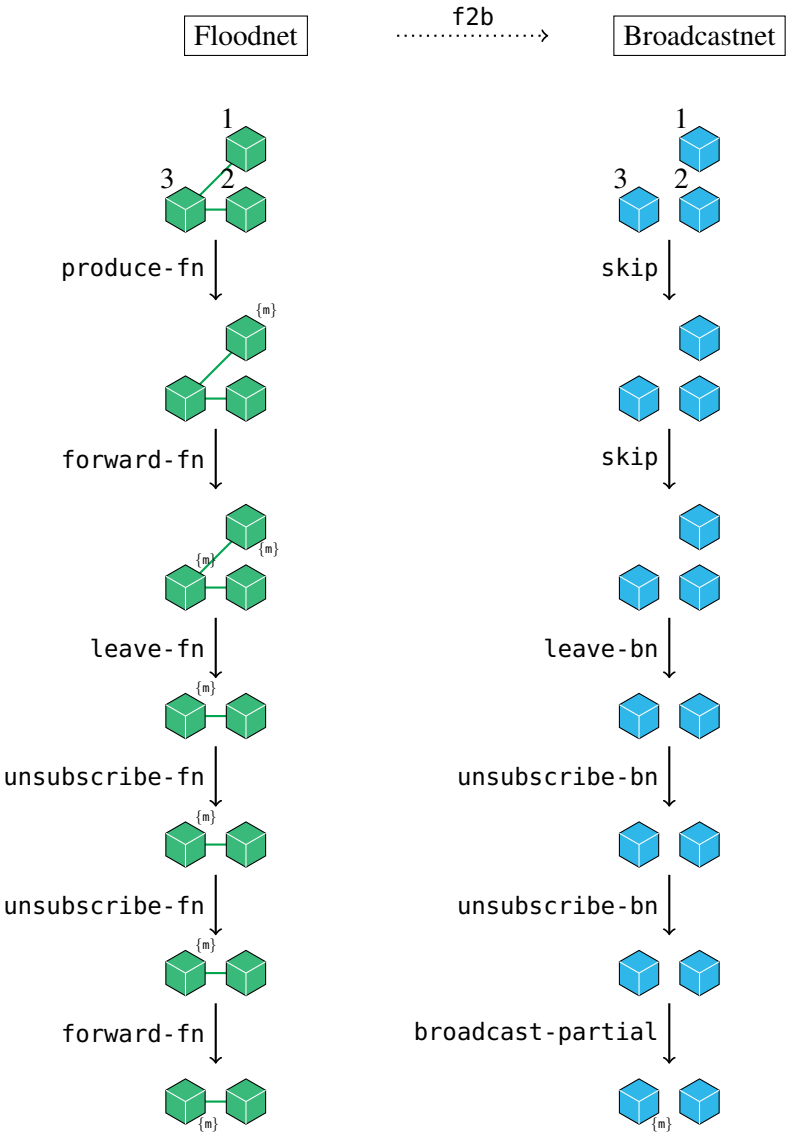

Figure 1: On the left is an example Floodnet trace. Broadcastnet states on the right are refinement maps of the Floodnet states on the left, and every step taken by the Broadcastnet states matches each step taken by the Floodnet states.

When we consider a static configuration where nodes do not change their subscriptions, no existing nodes are leaving, no new nodes are joining, and the network is connected in each topic, the recipients of the message `m` in `broadcast-partial` can be shown to be exactly those in `broadcast` *i.e.*, the subscribers of `(mssg-tp m)`. This aligns with Dijkstra's notion of self-stabilizing distributed systems [11], where the system is guaranteed to reach a legitimate configuration regardless of the initial state. In our case, once the system stabilizes (i.e., peer churn and subscription/unsubscription ceases),

the broadcast-partial step effectively becomes indistinguishable from a broadcast step, reinforcing the view that broadcast-partial is a generalization that accommodates transient perturbations while preserving desirable behavior in steady state.

```
(definecd rel-broadcast-partial-bn (s u :s-bn) :bool
  (^ (br-mssg-witness s u)
     (new-bn-mssgp (br-mssg-witness s u) s)
     (== u (broadcast-partial (br-mssg-witness s u)
                              (brd-receivers-bn (br-mssg-witness s u) u)
                              s))))

(definecd broadcast-partial (m :mssg ps :lop s :s-bn) :s-bn
  :ic (new-bn-mssgp m s)
  (broadcast-partial-help m ps s))

(definecd broadcast-partial-help (m :mssg ps :lop st :s-bn) :s-bn
  (match st
    (() nil)
    (((p . pst) . rst)
     (cons `(,p . ,(if (== p (car ps))
                       (mset :seen (insert-unique m (mget :seen pst)) pst)
                     pst))
           (broadcast-partial-help m (if (== p (car ps)) (cdr ps) ps) rst)))))
;; broadcast message receivers in a Broadcastnetwork
(definec brd-receivers-bn (m :mssg s :s-bn) :lop
  (match s
    (() ())
    (((p . pst) . rst) (if (in m (mget :seen pst))
                           (cons p (brd-receivers-bn m rst))
                         (brd-receivers-bn m rst)))))
```

We define rel-B, which holds for related states. rel-B is defined over a set of states combining both Floodnet and Broadcastnet states, which we define as borf. Notice that rel-B depends on Floodnet states satisfying the good-s-fnp predicate. This predicate ensures that each Floodnet state in the trace satisfies certain invariants. Given space constraints, we only list the invariant properties that hold true for good-s-fnp states at the end of the following listing.

```
(defdata borf (v s-bn s-fn))

(definec rel-B (x y :borf) :bool
  (v (rel-wf x y)
     (== x y)))

(definec rel-wf (x y :borf) :bool
  (^ (s-fnp x)
     (s-bnp y)
     (good-s-fnp x)
     (== y (f2b x))))

(definec rel-> (s u :borf) :bool
  (v (^ (s-fnp s) (s-fnp u) (good-rel-step-fn s u))
     (^ (s-bnp s) (s-bnp u) (rel-step-bn s u))))
```

```
(definec good-rel-step-fn (s u :s-fn) :bool
  (^ (good-s-fnp s)
     (good-s-fnp u)
     (rel-step-fn s u)))

;; A good-s-fnp state satisfies 2 predicates
(definec good-s-fnp (s :s-fn) :bool
  (^ (p!in-nsubs-s-fn s) (ordered-seenp s)))

;; Invariant 1: A peer p does not track its own subscriptions in the
;; :nsubs map. So, it can not forward a message to itself.
(propertyd prop=p!in-nsubs-s-fn (p :peer tp :topic s :s-fn)
  :h (^ (mget p s) (p!in-nsubs-s-fn s))
  :b (nin p (mget tp (mget :nsubs (mget p s)))))

;; Invariant 2: :seen components of Floodnet peers are ordered.
(property prop=ordered-seenp-cdar (s :s-fn)
  :h (^ s (ordered-seenp s))
  :b (orderedp (mget :seen (cdar s))))

(definec orderedp (x :tl) :bool
  (match x
    (() t)
    ((&) t)
    ((a . (b . &)) (^ (<< a b) (orderedp (cdr x))))))
```

Proving the WFS refinement requires proving the following three theorems: (i) WFS1 states that concrete Floodnet states are related to their corresponding Broadcastnet states under the refinement map (by relation `rel-B`), (ii) WFS2 states that the labelling function labels related states equally, and (iii) WFS3 states that given related states `s` and `w`, and given `s` steps to `u` under the transition relation, there exists a state, say `v`, such that `u` is matched by a step from `w` going to `v` such that `w` is related to `v`.

```
;; WFS1
(property b-maps-f2b (s :s-fn)
  :h (good-s-fnp s)
  :b (rel-B s (f2b s)))

;; WFS2. L is the labelling functions of our combined transition system
(definec L (s :borf) :borf
  (match s
    (:s-bn s)
    (:s-fn (f2b s))))

(property wfs2 (s w :borf)
  :h (rel-B s w)
  :b (== (L s) (L w)))

 ;; WFS3
(defun-sk exists-v-wfs (s u w)
  (exists (v)
    (^ (rel-> w v)
       (rel-B u v))))
```

```
(property wfs3 (s w u :borf)
  :h (^ (rel-B s w)
        (rel-> s u))
  :b (exists-v-wfs s u w))

;; Witness generating function for v
(definec exists-v (s u w :borf) :borf
  :ic (^ (rel-B s w)
         (rel-> s u))
  (if (null s)
      (if (null u)
          nil
        (exists-nil-v u))
    (exists-cons-v s u w)))

;; when w is nil
(definec exists-nil-v (u :borf) :borf
  :ic (^ u (rel-> nil u))
  (match u
    (:s-fn (exists-v1 nil u))
    (:s-bn u)))

;; when w is not nil
(definec exists-cons-v (s u w :borf) :borf
  :ic (^ s (rel-B s w) (rel-> s u))
  (cond
   ((^ (s-bnp s) (s-bnp w)) u)
   ((^ (s-fnp s) (s-bnp w)) (exists-v1 s u))
   ((^ (s-fnp s) (s-fnp w)) u)))

;; when s and u are Floodnet states
(definec exists-v1 (s u :s-fn) :s-bn
  :ic (good-s-fnp s)
  (cond
   ((rel-skip-fn s u) (f2b s))
   ((^ (rel-forward-fn s u)
       (!= (f2b s) (f2b u)))
    (broadcast-partial (br-mssg-witness (f2b s) (f2b u))
                       (brd-receivers-bn (br-mssg-witness (f2b s) (f2b u))
                                         (f2b u))
                       (f2b s)))
   (t (f2b u))))
```

4 Proof Organization

Given that we define our models using transition functions from states to states, whereas the refinement theorem is expressed in terms of transition relations, proving the monolithic refinement theorem can be a daunting task. In this section, we describe how we approached the mechanization of the refinement proof. The entire codebase can be logically partitioned into four stages:

- **State models and transition functions**: State models are described using `defdata`, and transition functions on the state models are described using `definec` and appear in files `bn-trx.lisp` and `fn-trx.lisp`. Apart from the input state, functions may accept additional arguments. For example `forward-fn` 2.2 accepts a peer `p` along with a message `m` that `p` forwards. These functions usually have input contracts to ensure that the extra arguments satisfy certain properties, for example, `p` should be a peer in the state `s`, and it should have `m` in its `pending` set of messages.
- **Properties of functions under the refinement map**: Given that we have defined functions that accept states and output states, we then prove theorems relating Floodnet states to their corresponding Broadcastnet states under the refinement map in file `f2b-commit.lisp`. For example, here is one such theorem:

```
(property prop=forward-fn (p :peer m :mssg s :s-fn)
 :h (^ (mget p s)
       (in m (mget :pending (mget p s)))
       (== (fn-pending-mssgs (forward-fn p m s))
           (fn-pending-mssgs s)))
 :b (== (f2b (forward-fn p m s))
        (f2b s)))
```

  Notice that these theorems still depends on variables that have not yet been skolemized, so as to ease the theorem proving process.
- **Transition relations**: We define the transitions relations for both Broadcastnet and Floodnet in `trx-rels.lisp`. Transition relations are boolean functions over two state variables, and hence, we also define witness functions for non-state variables appearing in the transition functions. In our running example, we instantiate `p` with `(find-forwarder s m)` and `m` could be any one of the pending messages in the state.
- **Combined states, transitions relations and correctness theorems**: Finally we prove the WFS theorems and their helper properties in `f2b-sim-ref.lisp`.

During development, some of the previous iterations of our model deviated from the metatheory. For example, in one iteration of the final theorem, the restrictions on the transition relations, which serve as guards against illegal behaviors, emerged as part of the hypotheses of WFS3. It forced us to understand the nature of good states, and to derive the required hypotheses from the invariants of the good states. In an another iteration of our models, the transition relations corresponding to each of the transitions a model can make, was augmented with natural numbers, such that transitions on the Floodnet states were matched by transitions on the Broadcastnet states bearing the same number. Eventually, this arrangement seemed unnecessary because even without the natural numbers, the theorem prover was able to pick the required transition based on theorems proved on them, and because of their definitions being disabled. Hence we would recommend to stick to the metatheory when implementing proofs of refinement, and always write the top level theorems, before embarking on proving lower-level theorems.

5 Related Work

Proof mechanization in context of P2P systems has been explored previously. Azmy et. al. [3] formally verified a safety property of Pastry, a P2P Distributed Hash Table (DHT), in TLA+ [22]. The safety property is that of correct delivery, which states that at any point in time, there is at most one node that answers a lookup request for a key, and this node must be the closest live node to that key. Their proof

assumes that nodes never fail, which is a likely event in any P2P system. Zave [41] utilized the Alloy tool [14] to produce counter-examples to show that no published version of Chord is correct w.r.t. the liveness property of the Chord ring-maintenance protocol: that the protocol can eventually repair all disruptions in the ring structure, given ample time and no further disruptions while it is working. Kumar et. al. modeled the Gossipsub [35] P2P protocol in ACL2s, formulated safety properties for its scoring function and showed using counter-examples that for some applications like Ethereum which configure Gossipsub in a particular way, it is possible for Sybil nodes to violate those properties, thereby creating large scale partition or eclipse attacks on the network [19, 18]. To the best of our knowledge, none of the previous work has attempted to prove the correctness of a P2P system by showing it as a refinement of a higher level specification.

There exist several provers to formally check properties of distributed systems, such as Dafny [13], TLA+, Ivy [32] and DistAlgo [34]. They operate by reducing a given specification to a decidable logic formula expressed entirely in First Order Logic. The basic tactic involves forming a conjunction of protocol invariants, invert it, and then using an SMT solver to (possibly) search for a counterexample. The issue in such systems is a lack of expressivity, which does not allow capturing properties over infinite traces. Another issue is that our models can be arbitrary, with nodes leaving and joining and with arbitrary pending messages in transit across a network. And we are reasoning about all possible behaviors of the protocol, which could not be done if we were to be limited to a decidable fragment of logic.

We wrote our models and proved our theorems in ACL2s. The ACL2 Sedan (ACL2s) [12, 6] is an extension of the ACL2 theorem prover[15, 16, 17]. On top of the capabilities of ACL2, ACL2s provides the following: (1) A powerful type system via the `defdata` data definition framework [9] and the `definec` and `property` forms, which support typed definitions and properties. (2) Counterexample generation capability via the `cgen` framework, which is based on the synergistic integration of theorem proving, type reasoning and testing [8, 10, 7]. (3) A powerful termination analysis based on calling-context graphs [29] and ordinals [26, 27, 28]. (4) An (optional) Eclipse IDE plugin [6]. (5) The ACL2s systems programming framework (ASPF) [40] which enables the development of tools in Common Lisp that use ACL2, ACL2s and Z3 as a service [38, 37, 20, 39].

6 Conclusions and Future Work

In this paper, we described our ACL2s models for Broadcastsub and Floodsub, Broadcastnet and Floodnet respectively, and proposed Broadcastnet as a specification of Floodnet. For both the models, we explained our transition systems (including state and transition relations) and design decisions. We described our refinement map `f2b`, the combined transition system and the equivalence relation `rel-B` relating related states. Finally we explained the refinement theorem.

In the future we would like to show that in a static configuration where a Floodnet is connected in each of the topics, a `forward-fn` transition can be matched by either a `broadcast-partial` or a `broadcast` transition. We would also like to refine Floodnet progressively until we approach a specification close to Gossipsub. By contrasting this lowest layer of our refinement chain to Gossipsub, we will be able to find and explain security issues in Gossipsub from a refinement point of view.

Acknowledgements We thank the anonymous reviewers for their thoughtful feedback and suggestions, which helped improve the quality and clarity of this work.

References

[1] *What is Publish/Subscribe*. https://docs.libp2p.io/concepts/pubsub/overview/. Accessed 12 May 2023.

[2] Ioannis Aekaterinidis & Peter Triantafillou (2018): *Peer-to-Peer Publish-Subscribe Systems*. In: *Encyclopedia of Database Systems, Second Edition*, doi:10.1007/978-1-4614-8265-9_1221.

[3] Noran Azmy, Stephan Merz & Christoph Weidenbach (2016): *A Rigorous Correctness Proof for Pastry*. In: *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, doi:10.1007/978-3-319-33600-8_5.

[4] Michael C. Browne, Edmund M. Clarke & Orna Grumberg (1988): *Characterizing Finite Kripke Structures in Propositional Temporal Logic*. doi:10.1016/0304-3975(88)90098-9.

[5] Miguel Castro, Peter Druschel, A-M Kermarrec & Antony IT Rowstron (2002): *SCRIBE: A large-scale and decentralized application-level multicast infrastructure*. *IEEE Journal on Selected Areas in communications*, doi:10.1109/JSAC.2002.803069.

[6] 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.

[7] Harsh Raju Chamarthi (2016): *Interactive Non-theorem Disproving*. Ph.D. thesis, Northeastern University, doi:10.17760/D20467205.

[8] Harsh Raju Chamarthi, Dillinger Peter C., Matt Kaufmann & Panagiotis Manolios (2011): *Integrating testing and interactive theorem proving*. doi:10.4204/EPTCS.70.1.

[9] Harsh Raju Chamarthi, Dillinger Peter C. & Panagiotis Manolios (2014): *Data Definitions in the ACL2 Sedan*. doi:10.4204/eptcs.152.3.

[10] 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.

[11] Edsger W. Dijkstra (1974): *Self-stabilizing Systems in Spite of Distributed Control*. *Commun. ACM*, doi:10.1145/361179.361202.

[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)*, doi:10.1016/j.entcs.2006.09.018.

[13] 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.

[14] Daniel Jackson (2019): *Alloy: a language and tool for exploring software designs*. doi:10.1145/3338843.

[15] Matt Kaufmann, Panagiotis Manolios & J Strother Moore (2000): *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, doi:10.1007/978-1-4615-4449-4.

[16] Matt Kaufmann, Panagiotis Manolios & J Strother Moore (2000): *Computer-Aided Reasoning: Case Studies*. Kluwer Academic Publishers, doi:10.1007/978-1-4757-3188-0.

[17] Matt Kaufmann & J Strother Moore (2022): *ACL2 homepage*. Available at https://www.cs.utexas.edu/users/moore/acl2/.

[18] Ankit Kumar, Max von Hippel, Panagiotis Manolios & Cristina Nita-Rotaru (2023): *Verification of GossipSub in ACL2s*. In: *International Workshop on the ACL2 Theorem Prover and Its Applications*, doi:10.4204/EPTCS.393.10.

[19] Ankit Kumar, Max von Hippel, Panagiotis Manolios & Cristina Nita-Rotaru (2024): *Formal Model-Driven Analysis of Resilience of GossipSub to Attacks from Misbehaving Peers*. In: *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*, doi:10.1109/SP54263.2024.00017.

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

[21] Ankit Kumar & Panagiotis Manolios (2025): *Proof of Refinement of Floodsub*. Available at `https://github.com/ankitku/FloodsubRef`. In submission.

[22] Leslie Lamport (2002): *Specifying systems: the TLA+ language and tools for hardware and software engineers*.

[23] Panagiotis Manolios (2001): *Mechanical Verification of Reactive Systems*. Ph.D. thesis, The University of Texas at Austin, Department of Computer Sciences, Austin TX.

[24] 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.

[25] 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.

[26] Panagiotis Manolios & Daron Vroon (2003): *Algorithms for Ordinal Arithmetic*. In: *Conference on Automated Deduction CADE*, doi:10.1007/978-3-540-45085-6_19.

[27] 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.

[28] Panagiotis Manolios & Daron Vroon (2005): *Ordinal Arithmetic: Algorithms and Mechanization*. *Journal of Automated Reasoning*, doi:10.1007/s10817-005-9023-9.

[29] Panagiotis Manolios & Daron Vroon (2006): *Termination Analysis with Calling Context Graphs*. In: *Computer Aided Verification CAV*, doi:10.1007/11817963_36.

[30] Pete Manolios (2023): *Reasoning About Programs*. Available at `https://www.ccs.neu.edu/home/pete/courses/Logic-and-Computation/2023-Fall/lectures.html`. Lecture notes, Northeastern University, CS 2800: Logic and Computation, Accessed 21 Apr 2025.

[31] Robin Milner (1971): *An Algebraic Definition of Simulation Between Programs*. In: *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*. Available at `http://ijcai.org/Proceedings/71/Papers/044.pdf`.

[32] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv & Sharon Shoham (2016): *Ivy: safety verification by interactive generalization*. doi:10.1145/2908080.2908118.

[33] Antony I. T. Rowstron & Peter Druschel (2001): *Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems*. In: *International Conference on Distributed Systems Platforms*, doi:10.1007/3-540-45518-3_18.

[34] Kumar Shivam, Vishnu Paladugu & Yanhong A. Liu (2023): *Specification and Runtime Checking of Derecho, A Protocol for Fast Replication for Cloud Services*. doi:10.1145/3584684.3597275.

[35] Dimitris Vyzovitis: *gossipsub: An extensible baseline pubsub protocol*. `https://github.com/libp2p/specs/blob/master/pubsub/gossipsub/README.md`. Accessed 28 Nov 2022.

[36] Dimitris Vyzovitis (2020): *GossipSub v1.0: An extensible baseline pubsub protocol*. `https://github.com/libp2p/specs/blob/master/pubsub/gossipsub/gossipsub-v1.0-old.md`. Accessed 23 Jan 2025.

[37] Andrew T. Walter, Benjamin Boskin, Seth Cooper & Panagiotis Manolios (2019): *Gamification of Loop-Invariant Discovery from Code*. In: *Proceedings of the Seventh AAAI Conference on Human Computation and Crowdsourcing, HCOMP*, doi:10.1609/HCOMP.V7I1.5277.

[38] Andrew T. Walter, David A. Greve & Panagiotis Manolios (2022): *Enumerative Data Types with Constraints*. In: *Formal Methods in Computer-Aided Design, FMCAD*, doi:10.34727/2022/ISBN.978-3-85448-053-2_-25.

[39] Andrew T. Walter, Ankit Kumar & Panagiotis Manolios (2023): *Proving Calculational Proofs Correct*. In: *Proceedings of the 18th International Workshop on the ACL2 Theorem Prover and Its Applications*, doi:10.4204/EPTCS.393.11.

[40] Andrew T. Walter & Panagiotis Manolios (2022): *ACL2s Systems Programming*. In: *Workshop on the ACL2 Theorem Prover and its Applications*, doi:10.4204/EPTCS.359.12.

[41] Pamela Zave (2012): *Using lightweight modeling to understand chord*. *Comput. Commun. Rev.*, doi:10.1145/2185376.2185383.

An ACL2s Interface to Z3

Andrew T. Walter Panagiotis Manolios
Khoury College
Northeastern University
Massachusetts, USA
walter.a@northeastern.edu p.manolios@northeastern.edu

We present Lisp-Z3, an extension to the ACL2s systems programming framework (ASPF) that supports the use of the Z3 satisfiability modulo theories (SMT) solver. Lisp-Z3 allows one to develop tools written using the full feature set of Common Lisp that can use both ACL2/s (either ACL2 or ACL2s) and Z3 as services, combining the power of SMT and interactive theorem proving. Lisp-Z3 is usable by anyone who would like to interact with Z3 from Common Lisp, as it does not depend on the availability of ACL2/s. We discuss the use of Lisp-Z3 in three applications. The first is a Sudoku solver. The second is SeqSolve, a string solver which solved a larger number of benchmark problems more quickly than any other existing solver at the time of its publishing. Finally, Lisp-Z3 was also used in the context of hardware-in-the-loop fuzzing of wireless routers, where low latency was an important goal. The latter two applications leveraged the ability of Lisp-Z3 to integrate Z3 with ACL2s code. We have further plans to use Lisp-Z3 inside of ACL2s to provide more powerful automated support for dependent types, and in particular more efficient generation of counterexamples to properties involving dependent types. This paper describes the usage and implementation of Lisp-Z3, as well as an evaluation of its use in the aforementioned applications.

1 Introduction

This paper describes a publicly available extension to our ACL2s systems programming framework [44] (ASPF) that supports the use of the Z3 satisfiability modulo theories (SMT) solver [35] as a service.

ASPF enables the development of tools that use ACL2 and ACL2s (the ACL2 Sedan) as a service by allowing one to write code that uses Common Lisp features that ACL2/s (ACL2 and ACL2s) restrict. This code can then interact with ACL2/s using a library provided by ASPF. We have used ASPF to build several systems, including a web-based loop invariant discovery game [41], a system for providing feedback for calculational proofs intended for pedagogical settings [43] and a system for automating the grading of homework involving different kinds of automata [27]. In our experience, ASPF particularly shines when building tools that are components of a larger system, especially when networking and foreign-function interfacing (FFI) are required.

Z3 is an SMT solver. This means that given a set of constraints within a supported theory, Z3 will attempt to determine whether or not that set of constraints is satisfiable. If so, Z3 can produce a satisfying assignment (a *model*) for the constraints. Z3 may be able to determine that the constraints are unsatisfiable as well, or it may instead exceed a timeout or resource limit and report that the satisfiability of the constraints is unknown. Lisp-Z3 provides an interface for expressing and asserting constraints, requesting that Z3 check the satisfiability of asserted constraints and accessing the produced satisfying assignment if Z3 determined that the constraints were satisfiable. The interface of Lisp-Z3 is intended to mirror the SMT-LIB2 [8] command interface as much as possible, making it especially easy to use for anyone who has experience with SMT-LIB2 (which Z3 and many other SMT solvers support). Many kinds of problems can be modeled using SMT, with a classic example being solving Sudoku puzzles.

Gamboa and Manolios (Eds):
ACL2 Workshop 2025
EPTCS 423, 2025, pp. 104–123, doi:10.4204/EPTCS.423.10

As prior work has reported [30, 36, 37], interactive theorem proving (ITP) and SMT are complementary techniques and their combination can be highly effective. The authors and their collaborators have found the combination of ACL2/s (which is an interactive theorem prover) and Z3 to be useful in multiple applications, including string solving (Kumar *et al.*'s TranSeq) [26] and security testing of wireless routers [42]. To support these applications, it was necessary to develop an ACL2/s or Common Lisp interface for Z3 with the right features—in the case of security testing, low latency was highly desirable, whereas incremental solving was important for the string solver. Existing interfaces did not fulfill these requirements. Our interface, which we call Lisp-Z3, consists of low-level bindings to Z3's C API as well as a higher-level interface on top to make it convenient to interact with Z3. Lisp-Z3 is usable by anyone who would like to interact with Z3 from Common Lisp, as it does not rely on functionality specific to ACL2/s. Nevertheless, we think of Lisp-Z3 as an extension of the ACL2s Systems Programming methodology, providing another reasoning backend in addition to ACL2/s.

In addition to using Lisp-Z3 when developing tools that use ACL2/s as a service, we are planning to use Lisp-Z3 to power functionality inside of ACL2s. In particular, we are working on Enumerative Data Types Modulo Theories, a generalization and extension of our wireless router security testing project [42] that aims to improve the ability of ACL2s to generate counterexamples in the presence of constraints. This will involve using Lisp-Z3 inside of ACL2s' cgen [13], which is integrated into the ACL2 waterfall.

The contributions of this work include: **(1)** A description of the design and implementation of Lisp-Z3, a major extension to ASPF that in addition to supporting ACL2s also supports Z3. With this extension, one can build tools that use both ACL2/s and Z3 as services, **(2)** A public release of the extended ASPF, including examples of the use of Lisp-Z3 in Common Lisp outside of ACL2/s, **(3)** an evaluation of the use of Lisp-Z3 in conjunction with the ASPF in three applications: a Sudoku solver, a state-of-the-art string solver and hardware-in-the-loop fuzzing of wireless routers.

The remainder of the paper is organized as follows: Section 2 gives a brief introduction to the interface of Lisp-Z3 through examples, Section 3 gives short introduction to Z3 and SMT-LIB2, Section 4 discusses how Lisp-Z3 is implemented, Section 5 walks through the development of a Sudoku solver using Lisp-Z3, Section 6 discusses the use of Lisp-Z3 in the SeqSolve string solver, Section 7 explores the use of Lisp-Z3 in hardware-in-the-loop fuzzing of wireless routers, Section 8 provides an overview of related work, and Section 9 concludes.

2 Usage

Listing 1 shows a basic example of the usage of Lisp-Z3. After initializing Z3, the variables `x` and `y` are declared in the same way that one might declare them in SMT-LIB2, using the `declare-const` command. `x` is declared to be a Boolean variable, and `y` is declared to be an integer variable. Next, the `z3-assert` function is used to add a constraint to Z3. The constraint added states that `x` must be true, and that `y` must be greater than or equal to 5. Note that this could be written as two independent calls to `z3-assert` rather than as a conjunction if desired. Next, the `check-sat` command is run, which asks Z3 to determine whether the conjunction of all of the constraints added to it is satisfiable. Here it will return `:SAT`, indicating that the constraints are indeed satisfiable. Finally, we call `get-model` to retrieve Z3's representation of a satisfying assignment to the free variables in the constraint we added. In this case, Z3's representation of one possible satisfying assignment is printed as follows:

```
#<Z3::MODEL
  X -> true
  Y -> 5    >
```

If one would like to interact with the assignment in Common Lisp, it is generally easier to instead call `get-model-as-assignment`, which will translate Z3's representation of the satisfying assignment into a Common Lisp list appropriate for use as let bindings. In this case, the output would be `((X T) (Y 5))`. Note that there are infinitely many satisfying assignments to this set of constraints, as `y` may be assigned any integer greater than 5. In principle Z3 could produce any of these assignments, though in this case it tends to generate the solution shown above, which is the satisfying assignment with the smallest possible value for `y`.

Listing 1: An example of a basic SMT query using Lisp-Z3, in the style of SMT-LIB2.

```
;; Load lisp-z3
(ql:quickload :lisp-z3)
;; Enter its package so we can use its functions without needing to
;; specify the z3 package.
(in-package :z3)
;; Set up Z3. Only needs to happen once, before other code that uses Z3
(solver-init)
;; Declare variables x and y
(declare-const x Bool)
(declare-const y Int)
;; Add an assertion
(z3-assert
  (and x (>= y 5)))
;; Check for satisfiability
(check-sat)
;; If satisfiable, get a satisfying assignment
(get-model)
```

Lisp-Z3 also allows one to declare variables inline with the `z3-assert` form. This is shown in Listing 2. This syntax is similar to that used by ACL2s' `definec` and `property` forms, making it easier for users familiar with those facilities to start using Lisp-Z3.

Listing 2: An example of a basic SMT query using Lisp-Z3, using inline declarations of variables rather than forward declarations as shown in Listing 1.

```
;; Set up Z3. Only needs to happen once, before other code that uses Z3
(solver-init)
;; Declare variables x and y and add an assertion over them
(z3-assert (x :bool y :int)
  (and x (>= y 5)))
;; Check for satisfiability
(check-sat)
;; If satisfiable, get a satisfying assignment and translate it into a
;; form that is usable as Common Lisp let bindings
(get-model-as-assignment)
```

It is important to note that the statement passed in to `z3-assert` to be asserted in Z3 will be interpreted using the semantics that Z3 assigns to the used operators. Z3's semantics for expressions diverge from the semantics of ACL2 in some cases, as will be discussed later.

3 Short Introduction to Z3 and SMT-LIB2

Z3 supports several input formats, but the default is SMT-LIB2 [8]. SMT-LIB2 was developed with the intention of creating a standard format for interacting with different SMT solvers. SMT-LIB2 consists of

several components, including a command language for use when interacting with a SMT solver. All of the languages that SMT-LIB2 provides are based on S-expressions. The base logic used in SMT-LIB2 is derived from many-sorted first-order logic with equality, meaning that functions, variables and operators have sorts associated with them. In this context, a sort can be thought of as a name for a type. SMT-LIB2 also provides a standard set of *theories*, each of which include declarations for the sorts and functions that the theory provides. For example, the `Ints` theory provides the `Int` sort and a set of functions over `Int`s (addition, multiplication, negation, subtraction, division, modulus, absolute value, and inequality relations).

To express a set of assertions and check its satisfiability using the SMT-LIB2 command format, one will generally do the following: 1) declare or define any sorts, functions and constants (variables) that will be used beyond what is provided by the theory in use; 2) manipulate the set of assertions maintained by the SMT solver, for example by adding assertions over the declared sorts, functions and constants; 3) request that the SMT solver perform a satisfiability check and print a model. The produced model may not have an interpretation (an assigned value) for every declared sort, function and constant from the assertion stack. This generally will occur if the satisfiability of the assertion stack is not dependent on that sort, function or constant having a particular value.

SMT-LIB2 solvers maintain an *assertion stack* that consists of *assertion levels*. Each assertion level is a set containing assertions as well as declarations of sorts, functions and constants. When the solver is asked to check satisfiability, it considers the contents of all of the assertion levels in the stack. SMT-LIB2 provides commands for manipulating the stack. `push` allows one to create a new assertion level, and `pop` removes the most recently introduced assertion level from the stack. This removes any of the assertions added since the popped assertion level was introduced. The behavior of popping on sort and variable declarations is controlled by the (`:global-declarations`) solver option. If this option is set to `false` (as is the default in Z3), a declaration of a sort or a variable is attached to the assertion level of the solver at the time of declaration. If that assertion level is popped off the stack, the declaration is removed. If the option is set to `true`, declarations of variables and sorts are unaffected by changes to assertion levels. Maintaining an assertion stack means that an SMT-LIB2 solver can support a kind of *incremental solving*, where satisfiability is queried multiple times, with modifications made to the set of assertions in between queries.

4 Implementation

Lisp-Z3 consists of two main parts: the low-level bindings to Z3's C API, and the higher-level interface that provides a convenient interface for asserting constraints and generally interacting with Z3. These two parts together make up an ASDF [1] system that can be loaded by many Common Lisp implementations.

4.1 The Low-Level Interface

Included in Z3's distribution is a library that can be used to integrate Z3 inside another program. Z3 provides APIs that allow one to call into this library from several different programming languages. We chose to write bindings for the C API provided by Z3, as C foreign function interfacing (FFI) is common and there is substantial support available for doing so in Common Lisp. We used the Common Foreign Function Interface (CFFI) library [2] to implement our bindings in a way that is portable across many Common Lisp implementations.

Interfacing with C in Common Lisp results in certain challenges. For example, to be able to call a

C function that takes in an argument of type `Z3_context`, the Common Lisp implementation needs to know the size of values of that type, the layout of any fields (if it is a C struct) and how to turn a Common Lisp value into a `Z3_context` value. Even just determining the size of the type is a complicated affair, as it generally requires looking at the C header files where the type is defined, which involves handling preprocessor directives which may appear in those header files, and then making a guess as to what size a C compiler would use for values satisfying that definition. In practice, FFI tools often manage these issues by generating a C file that includes the relevant types and interfacing with a C compiler to determine whatever information is needed about those types. For Lisp-Z3, we use CFFI's Groveller functionality. We provide a special Common Lisp file called a *Grovel file* that has a form for each Z3 type we would like to interact with. The Groveller evaluates this file to produce a C file which is then compiled and run. The result of running the resulting executable is another Lisp file that contains CFFI forms that describe the layout and size of the Z3 types we referenced. We can then load the Z3 library and use CFFI forms to create Lisp bindings for the Z3 functions that we would like to call, using the size and layout information that was gleaned previously.

The Grovel file must be aligned with the API provided by the version of Z3 running on the user's computer. For example, different versions of Z3 may provide different members for an enumeration type used to identify which built-in operator a function call is using. To make it easier for a user to generate an appropriate version of the Grovel file, we provide a Python script that will read Z3's C header files and generate a Grovel file appropriate for them. A similar issue exists for the file that contains bindings for each Z3 C function that we would like to expose, though we do not yet provide an automated way to generate that file. We try to ensure that Lisp-Z3 is shipped with files that should work with a relatively modern version of Z3. This is done by using the Grovel file generation script and manually removing or modifying functionality for maximal compatibility.

At this point, it is possible to call many of Z3's C API functions, but it is not convenient to do so. One needs to manually deal with memory management tasks, array types are a pain to deal with, printing values of Z3 types gives little useful information and the context value must be provided in practically every function call. An example highlighting the verbosity of the low-level interface is provided in Listing 3. Note that this example does not include any error handling and also avoids functionality that requires manual reference counting (memory management). This is where the high-level interface comes in!

Listing 3: An example highlighting the usage of the low-level interface.

```
;; The below form asserts the constraint (= (+ x y) 10) for integer variables
;; x and y, checks satisfiability and reports a satisfying assignment if SAT.
(let* ((ctx (z3-mk-context (z3-mk-config)))
       (slv (z3-mk-simple-solver ctx))
       (x (z3-mk-const ctx (z3-mk-string-symbol ctx "X") (z3-mk-int-sort ctx)))
       (y (z3-mk-const ctx (z3-mk-string-symbol ctx "Y") (z3-mk-int-sort ctx)))
       ;; add has arbitrary arity, so we need to provide the args in a temporary C array.
       (sum (with-foreign-array (arg-array z3-c-types::Z3_ast (list x y))
              (z3-mk-add ctx 2 arg-array)))
       (stmt (z3-mk-eq ctx sum (z3-mk-numeral ctx "10" (z3-mk-int-sort ctx)))))
  (z3-solver-assert ctx slv stmt)
  ;; Check whether the assertion is satisfiable
  (if (equal (z3-solver-check ctx slv) :L_TRUE)
      ;; SAT! Now we must get all of the constant interpretations (e.g. variable
      ;; assignments) from the model.
      (let ((model (z3-solver-get-model ctx slv)))
        (loop for i below (z3-model-get-num-consts ctx model)
```

```
              for decl = (z3-model-get-const-decl ctx model i)
              for name = (z3-get-symbol-string ctx (z3-get-decl-name ctx decl))
              for value-ast = (z3-model-get-const-interp ctx model decl)
              ;; Here we assume the value is a numeral and get it as a string
              collect (list name (z3-get-numeral-string ctx value-ast))))
       ;; Otherwise, UNSAT or unknown.
      'not-sat))
;; Outputs (("Y" "0") ("X" "10"))
```

4.2 The High-Level Interface

The high-level interface mitigates several of the pain points that the low-level interface gives rise to. It is written entirely in Common Lisp and uses the low-level interface internally to make calls to Z3.

The Context and Solver When interacting with Z3 programatically, one is nearly always doing so with respect to a particular *context* value. The context stores certain settings and global values as well as information needed for memory management (discussed later). Since most operations on Z3 types require the context that the value was created relative to, we define a wrapper type around each Z3 type that has a field for the relevant context in addition to the value itself. In addition to making it unnecessary for the user to pass a context value around when dealing with Lisp-Z3 code, this makes it possible to implement `describe-object` and `print-object` for each Z3 type, enabling Common Lisp to display useful printed representations for values of Z3 types.

Another important element of the Z3 C API is the *solver* value, which stores any constraints that the user adds and is needed when checking satisfiability or generating a satisfying assignment to the set of constraints. When Lisp-Z3 is initialized, a default solver is created and stored. This solver is used whenever the user does not specify one. Many parameters of the solver can be modified to control Z3's behavior—for example, one can set how many threads will be used by Z3 when checking satisfiability, the logic used to set up the SMT solver and the schedule used for performing restarts.

Memory Management As is often the case when interfacing with a C API from a language with automatic memory management, one must be careful to ensure that any allocated memory that passes over the language barrier is deallocated at an appropriate time. For many of the types that it defines, Z3's C API provides a manual reference counting interface for managing the lifetime of allocated memory. This means that each time we create an object that requires manual reference counting, like a solver, we must call a function to increment the reference counter for that object. This is implemented for each Z3 type by incrementing the reference counter in the initializer of the corresponding wrapper type. As long as an object's reference counter has a positive value, Z3 will not deallocate that object's memory. We use the trivial-garbage Common Lisp library [3] to attach a finalizer function to each such object that will run when the wrapper object has been garbage collected (*e.g.* when it is no longer referenced by any Lisp values). This finalizer function decrements the object's reference counter so that Z3 is notified that one fewer reference to the object exists. When the reference counter hits zero, Z3 can deallocate that object's memory.

Producing Expressions Lisp-Z3 aims to support expressing as many of the constraints that Z3 supports as possible. To assert constraints in a Z3 solver, we first need to convert them into Z3 AST objects. This

may seem trivial, since the default input format for the Z3 binary is based on S-expressions, but in practice it is more complicated than simply handing off an S-expression to Z3.

The primary mechanism that Lisp-Z3 provides for expressing constraints to be asserted in Z3 is the `z3-assert` macro. In addition to taking in an expression to be asserted, this macro can optionally take in a set of specifiers for free variables to be used in the assertion, as well as a solver object. Each specifier contains a name and a sort specification describing the signature of the variable. These will be described in more detail later. All assertions are performed with respect to a solver object. If the solver object is not provided explicitly, the default solver is used. The assertion is traversed recursively, with each argument of a function call or operator application being translated into a Z3 AST before the function call or operator application itself is translated. Whenever a reference to a free variable is found, an appropriate Z3 AST object referencing a free variable with the correct name and sort is created. Information about the set of known identifiers and their sorts is maintained by the solver. Lisp-Z3 has support for a subset of the operators supported by Z3. The operators can be referenced by the same name that they are known by in Z3's SMT-LIB2 interface, though some are known by additional names as well (*aliases*). A document describing the set of operators known by Lisp-Z3 is provided alongside its source code [40].

The way that SMT-LIB2 behaves in situations where there are multiple declarations of variables with the same name is different from the way that Common Lisp does. In particular, SMT-LIB2 provides a single namespace for variables (constants and functions) and allows multiple declarations of variables with the same name, given that they are associated with different sorts. To reference such a variable, it is necessary to disambiguate using the `as` form. For example, if both a constant of type `Int` and a function of type `(Int) -> String` have been declared with the name `x`, one must reference the constant using the form `(as x Int)`. An exception to this behavior is when variables are introduced by a form that introduces bound variables, like `forall` or `exists`. If such a form introduces a variable with name `x`, any references to `x` in the body of that form (unless inside another form that introduces `x` as a bound variable) will refer to the bound variable rather than any declaration outside of binding form.

To behave in a way that is more consistent with Common Lisp, Lisp-Z3 restricts the declarations of variables. In particular, Lisp-Z3 requires that at all times, any name is associated with at most one declared free variable. Declarations of variables are associated with the solver's assertion level at the time of the declaration and are removed when that assertion level is popped off the stack. This is consistent with the behavior specified by SMT-LIB2 when `:global-declarations` is `false`. To be clear, attempting to declare a variable with the same name and a different sort as one in the current assertion level or any assertion level below it will result in an error. The one exception is the introduction of bound variables, which behave in the same way that SMT-LIB2 describes above (any bindings with the same name as a bound variable are replaced in the context of the body of the form introducing the bound variables).

Our wrapper around the Z3 solver object contains an *environment stack* that maps identifiers to variable declarations at each assertion level. Lisp-Z3 provides two ways to introduce variable declarations: an ahead-of-time option (consistent with SMT-LIB2) and an inline option. The ahead-of-time option involves using the `declare-const` or `declare-fun` forms, which behave identically to the commands of the same name defined by SMT-LIB2. After checking that the variable is not already declared in the current assertion level or any level below it, a variable declared using either form is added to the solver's environment stack at the current assertion level. The variable can then be referred to in any assertions added at the current assertion level or above it. The inline option for declaring variables involves providing variable specifiers in a `z3-assert` form. These variable specifiers are processed to produce a mapping from each variable to a declaration, and the declarations are added to the solver's environment stack at the current assertion level. Just as with the ahead-of-time option, an attempt to declare a variable

with a name that is already mapped to a declaration but a different sort than that declaration will result in an error.

Using and Defining Sorts In SMT-LIB2, each sort is defined with some number of parameters (potentially zero). For example, the `Int` sort takes in zero parameters, and the `Seq` sort takes in a single parameter (representing the sort of the sequence's elements). A sort with at least one parameter is a *parametric* sort, while a sort with no parameters is a *non-parametric* sort. The name of a sort may be an *indexed identifier*, meaning that it is of the form `(_ <name> <idx1> ... <idxn>)`, where each `<idxi>` is either a number or a symbol. For example, a sort representing a bitvector of width 3 is represented using the following indexed identifier: `(_ BitVec 3)`. This is a non-parametric sort, though it may look like a parametric one.

So, why does this distinction exist between indexed identifiers and parametric ones? In SMT-LIB2, it is possible to define a *sort parameter*—a variable that ranges over sorts—and then to use that variable as a parameter for a parametric sort. Listing 4 shows an example where `X` is declared as a sort parameter, and then a variable `y` is declared to be a function from `Int` values to values of sort `X`. On the other hand, indexed identifiers are restricted so that the provided indices are literal values. This means that for any bitvector sort in a SMT-LIB2 query, the width of that sort is encoded syntactically, making it much easier to apply any analysis that might benefit from knowledge of the bitwidth.

Listing 4: An example highlighting how sort parameters can be expressed in SMT-LIB2 syntax.

```
;; X is a variable over sorts
(declare-sort-parameter X)
;; y is a variable over functions from Int to X
(declare-const y (-> Int X))
```

When declaring variables for use in assertions, it is necessary to provide sort specifiers to indicate what sort each variable should have. A sort specifier refers to either a non-parametric sort, a parametric sort, or a function rank. We will first discuss non-parametric and parametric sorts before discussing function ranks.

A sort specifier for a non-parametric sort is simply a symbol denoting the name of a non-parametric sort that is known to Lisp-Z3. The package of that symbol does not matter. For example, the SMT-LIB2 sort `Int` is known to Lisp-Z3, and both `:int` and `int` are sort specifiers denoting it. A sort specifier for a parametric sort is a list where the first element is a symbol indicating the name of the parametric sort and the remaining entries in the list are arguments for the parameters of the parametric sort. Different parametric sorts may take in different kinds of values for their parameters, including sorts. Parameters for such sorts can be provided as sort specifiers themselves. For example, `(:seq :int)` is a sort specifier that denotes the `(Seq Int)` sort in Z3.

SMT-LIB2 requires that each function have at least one *rank* associated with it [8]. A rank is a non-empty sequence of sorts, where the last sort is the return type of the function and the sequence of sorts up to the last sort (potentially empty) denotes the sorts of the parameters of the function. A sort specifier for a function rank consists of a list of the form `(:fn (<p1> ... <pn>) <r>)`, where each `<pi>` and `<r>` is a sort specifier for a parametric or non-parametric sort. In general a function may have multiple ranks, but Lisp-Z3 only supports free functions with a single rank, for similar reasons as it does not support variables that have the same name but different sorts. Function rank sort specifiers are processed by translating each of `<pi>` and `<r>` into Z3 sorts and then producing a Z3 function declaration object with the given name and rank.

When declaring variables inline using `z3-assert`, it is necessary to refer to the name of the variable's sort using a keyword symbol (which can be written as a symbol whose name starts with a colon). This is because the fact that a symbol is in the keyword package is used to identify that a particular entry in the variable specifiers for a `z3-assert` call refers to a type rather than a variable name. When using `declare-const` or `declare-fun`, the name of the sort will be normalized in such a way that the package that it is in is irrelevant.

Many sorts built-in to Z3, like `Int` and `Seq`, are available with the same names in Lisp-Z3. In addition, it is possible to define a subset of the user-defined sorts that Z3 allows. In particular, Lisp-Z3 supports enumeration sorts and tuple sorts. Enumeration sorts consist of a finite number of distinct constants. For example, one way to represent the value of a Sudoku square is as an enumeration sort containing only the integers between 1 and 9 inclusive. Such an enumeration sort can be defined in Lisp-Z3 using the `register-enum-sort` function, as shown in Listing 5.

Listing 5: An example of an enumeration sort being registered in Lisp-Z3.

```
(register-enum-sort :square (1 2 3 4 5 6 7 8 9))
```

Tuple sorts can be thought of like `struct` types in Common Lisp or record types in other languages. They consist of a set of fields, each of which has a name and a sort. The fields must have distinct names. An example of a definition of a tuple sort is shown in Listing 6.

Listing 6: An example of a tuple sort being registered in Lisp-Z3.

```
(register-tuple-sort :person ((age . :int) (name . :string)))
```

Both enumeration sorts and tuple sorts can be defined using the `declare-datatypes` SMT-LIB2 command, though that command allows for the definition of more complicated sorts than Lisp-Z3 does.

Interpreting Models When Z3 determines that the set of assertions loaded into the current solver is satisfiable, it is possible to request a *model* from Z3 that describes a satisfying assignment to the set of assertions. This model maps any free variables and sorts in the assertions to interpretations (values). However, Z3 may determine that the interpretation of a particular free variable does not impact the satisfiability of the assertions. The generated model will not provide an interpretation for such variables.

To be able to use the interpretations from a model in Common Lisp, it is necessary to translate them into Common Lisp values. This can be done by using the `(get-model-as-assignment)` function. The interpretations for constants are encoded as Z3 AST values, just like the AST values that we generate when producing constraints for asserting in Z3. For example, Z3 may encode an interpretation equivalent to the sequence `(1 2 3)` as a concatenation of unit sequences `(seq.++ (seq.unit 1) (seq.unit 2) (seq.unit 3))`. The code that translates these ASTs into Common Lisp values does not support all possible interpretations. Additionally, there are some ASTs that have multiple possible Common Lisp representations, or for which it is not possible to produce a perfectly identical Common Lisp representation using Z3's C API. These include algebraic number values representing irrational roots of polynomials (for example, $\sqrt{2}$)—Lisp-Z3 will by default translate such values into a Common Lisp floating-point value, losing some precision.

Function interpretations are particularly interesting to look at. Z3 represents an interpretation for a function f using a combination of a map from function inputs to outputs M_f and a default value $else_f$. The value of the function on a particular set of arguments $f(a_1,...,a_n)$ is either the value that the set of arguments maps to in M_f (if that set of arguments is mapped by M_f) or the default value

$else_f$ otherwise. By default, Lisp-Z3 will translate a function interpretation for a function f into an S-expression containing a map with all of the entries from M_f but where each argument value and output value has been transformed into a Common Lisp value, plus a designated `:default` key that is mapped to $else_f$ transformed into a Common Lisp value.

As a result of the restrictions that Lisp-Z3 imposes on variable declarations, it is always the case that a model produced by Z3 will contain at most one interpretation for each variable name. This makes it possible to unambiguously interpret the assignment produced by `(get-model-as-assignment)`, as otherwise it would be possible for multiple variables for the same name but different sorts to be included in the assignment. On the other hand, it is possible that some of the variables that were constrained will not appear in the assignment produced by `(get-model-as-assignment)`. This occurs when Z3 does not assign the variable an interpretation in the produced model.

Another way to interact with the model produced by Z3 is to use the `eval-under-model` form provided by Lisp-Z3. This form takes in a expression to be converted into a Z3 AST in the same way that `z3-assert` does and evaluates it under either the given model or the result of `(get-model)` if no model is provided. The result of the evaluation is another Z3 AST, which is converted into a Common Lisp value and returned. By default `eval-under-model` will ask Z3 to perform completion on the given model when evaluating the given expression, meaning that if the statement to evaluate references a variable that was used in the assertion stack but that is not assigned an interpretation in the given model, Z3 will assign that variable some value that satisfies its sort (and it will use this value consistently if the variable appears more than once in the statement to evaluate).

Additional Features Z3 provides a wide variety of features, many of which have not been discussed so far. Lisp-Z3 supports Z3's optimization functionality, allowing one to specify objective functions to maximize or minimize as well as add soft constraints. It is possible to access statistics that Z3 gathers during the solving process, something which is often helpful when trying to understand Z3's performance on a particular set of assertions.

5 Sudoku

A classic example of a puzzle that can be encoded as an SMT problem is Sudoku. A traditional Sudoku puzzle consists of a 9x9 grid of squares (a *grid*), where each square is either blank or contains an integer between 1 and 9 inclusive. The grid is partitioned into nine 3x3 submatrices (*subgrids*). The goal of the player is to fill in any blank squares such that the resulting grid satisfies the following: for each row, column and subgrid, that group of squares must contain distinct values. A well-formed Sudoku puzzle has a unique solution [16]. Figure 1 shows an example of a Sudoku puzzle and its solution.

One can encode Sudoku as an SMT problem using a variety of different representations, but here we show one: representing the value of each square using an integer variable, appropriately constrained to be between 1 and 9 inclusive. Listing 7 contains an implementation of a Sudoku solver using Lisp-Z3. It consists of Common Lisp functions that generate the needed constraints for each square, row, column and subgrid, which can be asserted in Z3 once when the program starts up. There is also a function that translates a representation of a Sudoku problem (a "starting grid") into a set of equality constraints that represent the values of non-blank squares given in the problem, which can then be asserted in Z3. This example highlights Z3's scope functionality by first asserting the constraints that are constant across all Sudoku problems (the square, row, column and subgrid constraints), and then creating a new scope before adding the constraints for a particular Sudoku problem. This new scope can then be exited after

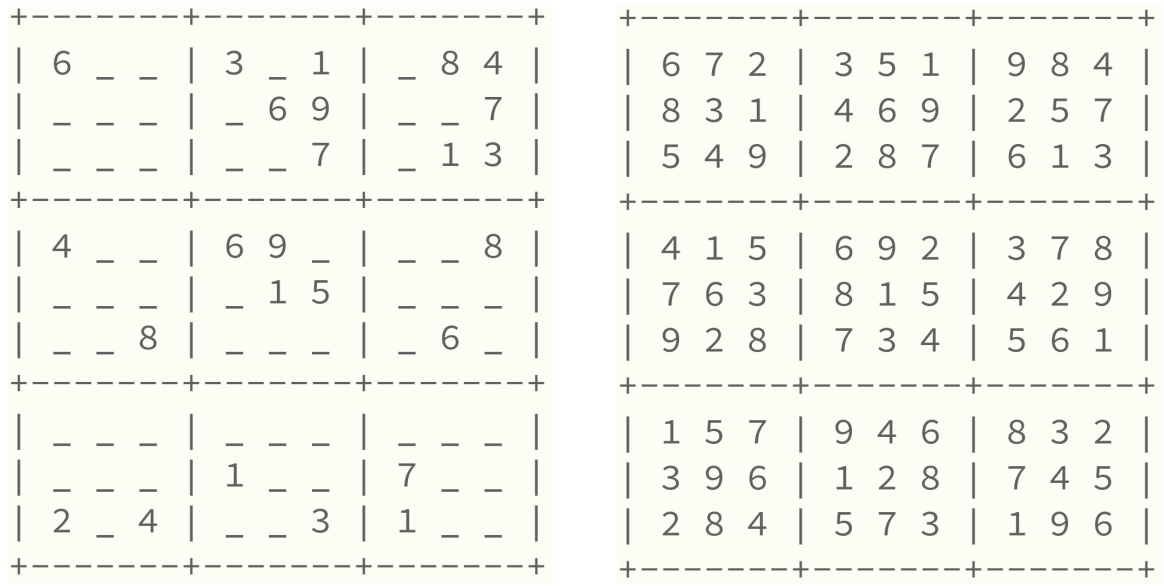

Figure 1: A Sudoku puzzle (left) and its solution (right). _ is used to denote a blank square.

the Sudoku problem is solved so that the Z3 solver can be used again for another Sudoku problem without needing to re-assert the initial set of constraints.

Listing 7: An implementation of a Sudoku solver using Lisp-Z3. This is an excerpt from the Sudoku example that is provided with Lisp-Z3, which also includes pretty-printing and several Sudoku puzzles.

```
;; Turn an index into a Sudoku grid into the variable corresponding to that square's value.
(defun idx-to-cell-var (idx)
  (intern (concatenate 'string "C" (write-to-string idx))))

;; We'll encode the sudoku grid in the simplest way possible, 81 integers
(defconstant +cell-vars+
  (loop for idx below 81
        append (list (idx-to-cell-var idx) :int)))

;; We limit the integers to values between 1 and 9, inclusive
(defconstant cell-range-constraints
  (loop for idx below 81
        append `((<= 1 ,(idx-to-cell-var idx))
                 (>= 9 ,(idx-to-cell-var idx)))))

;; distinct is a built-in Z3 function that is true iff none of its arguments are equal.

;; The values in each row must be distinct
(defconstant row-distinct-constraints
  (loop for row below 9
        collect `(distinct
                  ,@(loop for col below 9
                          collect (idx-to-cell-var (+ (* 9 row) col))))))

;; The values in each column must be distinct
(defconstant col-distinct-constraints
  (loop for col below 9
```

```
        collect `(distinct
                  ,@(loop for row below 9
                          collect (idx-to-cell-var (+ (* 9 row) col))))))

;; The values in each 3x3 box must be distinct
(defconstant box-distinct-constraints
  ;; These numbers are the indices of the top-left square of each box
  (loop for box-start in '(0 3 6 27 30 33 54 57 60)
        collect `(distinct
                  ;; These numbers are the offsets of each square in a
                  ;; box from the index of the box's top-left square
                  ,@(loop for box-offset in '(0 1 2 9 10 11 18 19 20)
                          collect (idx-to-cell-var (+ box-start box-offset))))))

;; Set up the initial constraints on the grid
(defun init ()
  (solver-init)
  ;; z3-assert-fn allows us to assert an expression generated by executing some Common Lisp code.
  (z3-assert-fn +cell-vars+ (cons 'and cell-range-constraints))
  (z3-assert-fn +cell-vars+ (cons 'and row-distinct-constraints))
  (z3-assert-fn +cell-vars+ (cons 'and col-distinct-constraints))
  (z3-assert-fn +cell-vars+ (cons 'and box-distinct-constraints)))

;; This generates constraints based on a "starting grid".
;; This starting grid is a length-81 list representation of the 9x9 Sudoku grid in row-major order.
;; The list should have a _ in cells where no initial value is given.
(defun input-grid-constraints (grid)
  (loop for entry in grid
        for idx below 81
        when (not (equal entry '_))
        collect `(= ,(idx-to-cell-var idx) ,entry)))

(defun solve-grid (input-grid)
  (solver-push)
  (z3-assert-fn +cell-vars+ (cons 'and (input-grid-constraints input-grid)))
  (let* ((sat-res (check-sat))
         (res (if (equal sat-res :sat)
                  (get-model-as-assignment)
                sat-res)))
    (progn (solver-pop)
           res)))

;; Now, use the solver! We assume the Sudoku grid from Figure 1 is loaded in *fig1-grid*.
(init)
(solve-grid *fig1-grid*)
```

Generating constraints programatically makes it easy to experiment with variants of Sudoku that have different-sized grids. The traditional Sudoku game seen above can be thought of as 3x3 Sudoku—each subgrid is contains 3 rows and 3 columns of squares and the top-level grid contains 3 rows and 3 columns of subgrids. The Lisp-Z3 examples contain code for a solver that can solve nxn Sudoku problems.

Note that the above Sudoku solver does not make use of any ACL2/s functionality. However, one could imagine using this Sudoku solver implementation as an oracle for solutions to a Sudoku problem inside of ACL2/s. Its output need not be trusted—instead one could write ACL2/s functions to validate that the produced solution is indeed a valid solution and matches with the given Sudoku problem.

6 Application: String Solving

Several applications benefit from the ability to perform satisfiability checking over string equations ("string solving"). These include security analysis [19, 20], program verification [25, 10] and type checking [38]. Many string solvers exist, including Z3str3 [9] which is part of Z3.

Kumar and Manolios used Lisp-Z3 in their string solver SeqSolve [26]. In addition to supporting string equation constraints, SeqSolve allows one to express LIA constraints over the lengths of string variables. As part of its solving process, SeqSolve generates LIA constraints over the lengths of string variables and uses Lisp-Z3 to determine whether or not these constraints are satisfiable. SeqSolve uses Z3's incremental solving capabilities (*e.g.* the assertion stack) to manage adding and removing constraints as appropriate as the string solving algorithm partitions the search space and explores each partition. SeqSolve is a particularly good example of the advantages of ASPF, as it is partially written in ACL2s, enabling it to take advantage of defdata data definitions and to express guarantees regarding those functions written in ACL2s. This includes guarantees about types (the function always returns a value satisfying its signature if it was called with arguments satisfying its signature) as well as regarding termination (unless explicitly configured not to, ACL2/s requires that any admitted function terminates).

Kumar and Manolios evaluated SeqSolve on a set of benchmarks, comparing against a set of string solvers that also supported length constraints. The benchmark set was derived mainly from benchmarks used in prior work, omitting benchmarks outside of the theory that SeqSolve supports. Kumar and Manolios found that SeqSolve solved a larger number of the benchmark problems in a shorter time than any of the other string solvers at the time of their paper's publishing. The results of Kumar and Manolios' work [26] highlight the ability of Lisp-Z3 to enable the use of Z3 alongside ACL2/s in a performant way, and their feedback was invaluable in guiding continued improvements to Lisp-Z3.

7 Application: Wi-Fi Fuzzing

Wireless communication protocols are ubiquitous in modern life, connecting devices ranging from smartphones to medical implants to the Internet. One of the most prevalent wireless communication protocols is the IEEE 802.11 Wi-Fi protocol [4]. Given the reach and impact of devices implementing Wi-Fi and the complexity of the protocol, it is important to evaluate the correctness and security of Wi-Fi infrastructure that will be used in sensitive applications. It is for this reason that we collaborated with a group at Collins to work on hardware-in-the-loop fuzzing of Wi-Fi routers [42].

Fuzzing is a technique for software testing wherein the system under test (SUT) is run on generated inputs with the goal of evaluating its reliability. Hardware-in-the-loop fuzzing involves using fuzzing to test a physical hardware device. This introduces several challenges above and beyond software fuzzing, including the introduction of timing constraints. These challenges, in conjunction with the complexity of the Wi-Fi protocol itself, mean that fuzzing Wi-Fi devices is challenging. Our work focused primarily on a particular part of the 802.11 specification—the probe request frame, which is used to request information about a Wi-Fi router prior to connecting to it. A 802.11 frame contains several fields, one of which is a body that itself consists of a sequence of elements. These elements each contain an ID indicating the kind of the element (are chosen from a set of possible element kinds) and a body, the valid values of which are determined by the element's kind.

Our collaborators had been working on model-based fuzzing of Wi-Fi routers, focusing on the probe request frame. They wrote a model expressing the structure of and constraints on a probe request frame and (through some tooling) translated the model into a SMT query, a satisfying assignment for which

corresponded to a valid probe request frame. They then used an approach called trapezoidal generalization [21] to generate a large number of different probe request frames from a single satisfying assignment. This was necessary because solving the SMT query was quite slow. The probe request frames were then sent to the SUT, which was monitored for availability.

We were interested in comparing our collaborators' approach for model-based fuzzing against one using ACL2s' enumeration facilities. To do this, we translated our collaborators' model into data definitions in ACL2s using the defdata system. By doing so, we were able to make use of the *enumerators* that ACL2s generates for each defdata type. The enumerator for a type is a function that maps natural numbers to elements of the type, making it possible to generate arbitrary elements of the type. This is used inside of ACL2s to support counterexample generation and automated testing.

An important factor for the success of model-based fuzzing is the ability to explore a large swathe of the space of possible input for the SUT (in the case of a Wi-Fi router, the space of possible frames). One relevant parameter in the case of probe request frames is the size of the generated frames, computed by summing the size of each element in the frame's body with the size of the frame's header. There exist valid probe request frames with any size between 172 and 2741 bytes (inclusive). It is possible that a bug in a Wi-Fi router's handling of probe request frames may only occur given very large or small frames. Therefore, we evaluated the ability of ACL2s to generate probe request frames with a particular size against an SMT approach. We found that both struggled to generate probe request frames across the entire range of valid sizes. We determined that the reason why ACL2s struggled to generate frames with a particular size was because it struggled to reason about the size of the different parts of a frame independently from their contents and pass that information to the frame's enumerator.

To produce a more performant approach, we split the task of generating a frame with a particular size into two steps: we solve the size constraints first (in Z3) and then pass that information along to the appropriate enumerators in ACL2s. This leverages the strengths of each tool—Z3 is highly capable at solving constraints involving linear integer arithmetic, and ACL2s makes it easy to specify types describing the body of each element, as the constraints on the contents of the body can be difficult for Z3 to handle.

To evaluate the performance of our approach (which we will refer to as ACL2s-ETC), we compared it against an approach that purely used ACL2s' counterexample generation (referred to as ACL2s-ET) and one that purely used Z3. We already had the ACL2s version of the model and we translated the model into SMT-LIB2 constraints for use in evaluating the pure-Z3 approach. We then measured the throughput of the three approaches when queried for frames with a particular length. We evaluated across lengths from 0 to 5000 bytes, in increments of 10 bytes. Given the model, frames exist with sizes between 172 and 2741 bytes inclusive, so the evaluated range contained both satisfiable and unsatisfiable queries. The results are shown in Figure 2 (note that results for frame sizes between 4000 and 5000 bytes are elided).

The results show a clear win for ACL2s-ETC: it was able to generate frames more quickly than either ACL2s-ET or Z3 across the entire range of evaluated frame sizes, and much more quickly (nearly two orders of magnitude) across the range of satisfiable frame sizes. These results are strong evidence for the effectiveness of the combination of ACL2s and Z3 and of the ability of Lisp-Z3 to support such a combination. Further discussion of these results is available in our FMCAD paper [42].

7.1 Enumerative Data Types Modulo Theories

Our evaluation above showed the effectiveness of our ACL2s-ETC approach in a particular application, but implementing our approach there required a fair amount of manual effort with respect to plumbing together defdata enumerators and Z3. We would like to generalize our approach in such a way that users

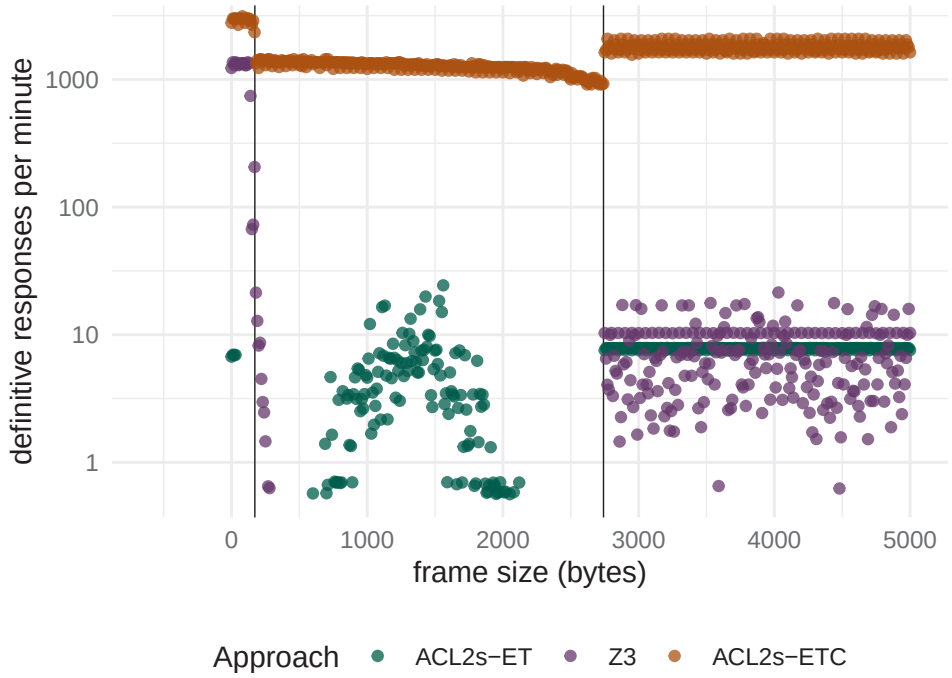

Figure 2: The number of frames generated per minute using each of three approaches when queried for frames with a given length. Z3 denotes an implementation that uses Z3 and an SMT-LIB2 version of the model to generate frames, ACL2s-ET uses ACL2s' counterexample generation and ACL2s-ETC uses a combination of Z3 and ACL2s. Only instances where the model returned a definitive response (*e.g.* not "unknown" or "timeout") are shown. The two vertical lines represent the minimum frame size and the maximum frame size; any responses outside of that range were all UNSAT, and any within that range were SAT. This figure is similar to Figure 7 in [42].

can take advantage of it in a highly automated fashion. This is the main idea behind our ongoing work on *enumerative data types modulo theories* (EDT), an extension of defdata that allows one to express what we call *parameters* of data types—features that can be constrained and solved for using something like Z3. For example, in our Wi-Fi fuzzing work we would define a frame size parameter which could then be solved for ahead of time if the user wrote a constraint over its value. We hope to use Lisp-Z3 to power our implementation of EDT inside of ACL2s' cgen counterexample generation functionality.

8 Related Work

The ACL2 Sedan (ACL2s) [17, 11] is an extension of the ACL2 theorem prover[22, 23, 24]. On top of the capabilities of ACL2, ACL2s provides the following: (1) A powerful type system via the defdata data definition framework [14] and the `definec` and `property` forms, which support typed definitions and properties. (2) Counterexample generation capability via the cgen framework, which is based on the synergistic integration of theorem proving, type reasoning and testing [13, 15, 12]. (3) A powerful termination analysis based on calling-context graphs [34] and ordinals [31, 32, 33]. (4) An (optional) Eclipse IDE plugin [11]. (5) The ACL2s systems programming framework (ASPF) [44] which enables

the development of tools in Common Lisp that use ACL2, ACL2s and Z3 as a service [42, 41, 26, 43].

The most directly relevant existing system to Lisp-Z3 is the Smtlink tool, developed by Peng and Greenstreet [36, 37]. In short, Smtlink allows one to use SMT to discharge ACL2 goals in a way that only involves trusting a small amount of code (in addition to ACL2 and Z3). In more detail, Smtlink provides a set of verified clause processors for transforming ACL2 goals into equivalent forms that are better suited for the SMT solver and a trusted clause processor that translates an ACL2 goal into a set of constraints for Z3, calls Z3, and interprets the output. Smtlink provides support for reporting counterexamples if reported by Z3 as a result of a failed proof.

Part of the challenge of implementing a system like Smtlink is that of providing a translation of an ACL2 form into Z3 that accounts for the differing semantics of ACL2 and Z3. This is especially true given that ACL2 uses an untyped logic and Z3 uses a many-sorted logic. For example, consider the ACL2 theorem in Listing 8 which can be proven to hold.

Listing 8: A theorem expressed using ACL2s' property form. Note that for ACL2s to accept this property, the contract checking feature of `property` must be disabled. [1]

```
(property (x :bool y :int)
  (= (+ x y) y))
```

This is true in ACL2 since ACL2's arithmetic functions generally treat any non-number arguments as though they were 0. However, the corresponding Z3 expression is falsifiable! This is shown in Listing 9. The reason for this is that Z3 happens to treat `true` as 1 and `false` as 0 in the context of arithmetic operators [2].

Listing 9: A naïve translation of the theorem in Listing 8 into Lisp-Z3 calls.

```
;; We negate the statement that we are trying to prove, and if
;; Z3 determines UNSAT then the statement is valid.
(z3-assert (x :bool y :int)
  (not (= (+ x y) y)))
(check-sat) ;; returns :sat, therefore the statement is not valid.
```

Since Smtlink allows one to use the fact that Z3 can prove a transformed ACL2 proof obligation to justify the correctness of the obligation in ACL2, a difference in semantics between the two that is not accounted for can load to unsoundness. The authors of Smtlink took particular care to ensure that their translation between ACL2 and Z3 expressions preserve validity. We chose to develop a lighter-weight solution in Lisp-Z3, insofar as it does not provide the ability to translate of an ACL2 expression into a Z3 expression in a way that preserves validity.

Manolios and Srinivasan were early advocates for integrating decision procedures into interactive theorem provers, suggesting an integration of the UCLID decision procedure with ACL2 in 2004 [28]. This integration was later performed and used to verify pipelined processor models [29, 30], enabling the automated verification of proofs that neither UCLID nor ACL2 could handle alone. Srinivasan went on to develop an integration of the Yices SMT solver with ACL2 as part of his PhD thesis [39]. Other researchers have investigated the integration of SMT into the Isabelle/HOL [18] and Coq [5] theorem provers, and work is ongoing to integrate the CVC5 SMT solver [7] and the Lean theorem prover [6].

[1] Contract checking can be disabled by evaluating `(set-acl2s-property-table-test-contracts? nil)` and `(set-acl2s-property-table-check-contracts? nil)`

[2] Note that this behavior is an extension of the behavior that SMT-LIB requires, and other SMT solvers supporting SMT-LIB may handle things differently. For example, CVC5 reports an error when trying to add that assertion, as it does not define the addition and multiplication operators on Booleans.

9 Conclusion and Future Work

We presented Lisp-Z3, an extension to the ACL2s systems programming framework (ASPF) that supports the use of Z3 as a service. The source code for Lisp-Z3 plus documentation and several examples of its usage are publicly available [40]. We also discussed three applications of our extended ASPF, the first being a Sudoku solver and the second being the SeqSolve string solver. The last application involved testing of wireless routers, where using a combination of ACL2s and Z3 resulted in substantially improved performance over pure-ACL2s and pure-Z3 approaches. We expect to use Lisp-Z3 inside of ACL2s as part of our ongoing work on enumerative data types modulo theories.

There are many improvements that we would like to make to Lisp-Z3. These include supporting a larger subset of the commands, operators and sorts that Z3 and SMT-LIB2 provide, developing an optional integration between Z3 sorts and ACL2s defdata types and enabling the use of other SMT solvers on the backend (in particular, CVC5). We are interested in getting more feedback from external users of the interface and encourage anyone interested in using Lisp-Z3 to experiment with it and reach out with any questions, comments or feedback.

Acknowledgments Lisp-Z3 has been greatly improved thanks to feedback from its users, including Ankit Kumar and the students in the Fall 2021 and 2022 sections of CS4820 at Northeastern University. Additionally, we would like to thank David Greve, who collaborated on Wi-Fi fuzzing with us at Collins, as well as Konrad Slind, Kristopher Cory and all of the other folks at Collins who we worked with.

References

[1] *ASDF - Another System Definition Facility*. Available at `https://asdf.common-lisp.dev/`.

[2] *CFFI - The Common Foreign Function Interface*. Available at `https://cffi.common-lisp.dev/`.

[3] *Trivial Garbage*. Available at `https://trivial-garbage.common-lisp.dev/`.

[4] (2021): *IEEE Standard for Information Technology–Telecommunications and Information Exchange between Systems - Local and Metropolitan Area Networks–Specific Requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*. *IEEE Std 802.11-2020 (Revision of IEEE Std 802.11-2016)*, pp. 1–4379, doi:10.1109/IEEESTD.2021.9363693.

[5] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry & Benjamin Werner (2011): *A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses*. In Jean-Pierre Jouannaud & Zhong Shao, editors: *Certified Programs and Proofs - First International Conference, CPP 2011. Proceedings*, *Lecture Notes in Computer Science* 7086, Springer, pp. 135–150, doi:10.1007/978-3-642-25379-9_12.

[6] Haniel Barbosa (2023): *Challenges in SMT Proof Production and Checking for Arithmetic Reasoning (Invited Paper)*. In Erika Ábrahám & Thomas Sturm, editors: *Proceedings of the 8th SC-Square Workshop co-located with the 48th International Symposium on Symbolic and Algebraic Computation, SC-Square@ISSAC 2023*, *CEUR Workshop Proceedings* 3455, CEUR-WS.org, pp. 1–9. Available at `https://ceur-ws.org/Vol-3455/invited1.pdf`.

[7] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli & Yoni Zohar (2022): *cvc5: A Versatile and Industrial-Strength SMT Solver*. In Dana Fisman & Grigore Rosu, editors: *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Proceedings, Part I*, *Lecture Notes in Computer Science* 13243, Springer, pp. 415–442, doi:10.1007/978-3-030-99524-9_24.

[8] Clark Barrett, Pascal Fontaine & Cesare Tinelli (2025): *The SMT-LIB Standard: Version 2.7*. Technical Report, Department of Computer Science, The University of Iowa. Available at `www.SMT-LIB.org`.

[9] Murphy Berzish, Vijay Ganesh & Yunhui Zheng (2017): *Z3str3: A string solver with theory-aware heuristics*. In Daryl Stewart & Georg Weissenbacher, editors: *2017 Formal Methods in Computer Aided Design, FMCAD 2017*, IEEE, pp. 55–59, doi:10.23919/FMCAD.2017.8102241.

[10] Tevfik Bultan, Fang Yu, Muath Alkhalaf & Abdulbaki Aydin (2017): *String Analysis for Software Verification and Security*. Springer, doi:10.1007/978-3-319-68670-7.

[11] 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.

[12] Harsh Raju Chamarthi (2016): *Interactive Non-theorem Disproving*. Ph.D. thesis, Northeastern University, doi:10.17760/D20467205.

[13] 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.

[14] 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*, doi:10.4204/EPTCS.152.3.

[15] 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`.

[16] Jean-Paul Delahaye (2006): *The science behind Sudoku*. *Scientific American* 294(6), pp. 80–87, doi:10.1038/scientificamerican0606-80.

[17] 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.

[18] Pascal Fontaine, Jean-Yves Marion, Stephan Merz, Leonor Prensa Nieto & Alwen Fernanto Tiu (2006): *Expressiveness + Automation + Soundness: Towards Combining SMT Solvers and Interactive Proof Assistants*. In Holger Hermanns & Jens Palsberg, editors: *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Proceedings*, *Lecture Notes in Computer Science* 3920, Springer, pp. 167–181, doi:10.1007/11691372_11.

[19] Xiang Fu & Chung-Chih Li (2010): *A String Constraint Solver for Detecting Web Application Vulnerability*. In: *Proceedings of the 22nd International Conference on Software Engineering & Knowledge Engineering (SEKE'2010)*, Knowledge Systems Institute Graduate School, pp. 535–542.

[20] Vijay Ganesh, Adam Kiezun, Shay Artzi, Philip J. Guo, Pieter Hooimeijer & Michael D. Ernst (2011): *HAMPI: A String Solver for Testing, Analysis and Vulnerability Detection*. In Ganesh Gopalakrishnan & Shaz Qadeer, editors: *Computer Aided Verification - 23rd International Conference, CAV 2011. Proceedings*, *Lecture Notes in Computer Science* 6806, Springer, pp. 1–19, doi:10.1007/978-3-642-22110-1_1.

[21] David A. Greve & Andrew Gacek (2018): *Trapezoidal Generalization over Linear Constraints*. In Shilpi Goel & Matt Kaufmann, editors: *Proceedings of the 15th International Workshop on the ACL2 Theorem Prover and Its Applications*, *EPTCS* 280, pp. 30–46, doi:10.4204/EPTCS.280.3.

[22] Matt Kaufmann, Panagiotis Manolios & J Strother Moore (2000): *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, doi:10.1007/978-1-4615-4449-4.

[23] Matt Kaufmann, Panagiotis Manolios & J Strother Moore (2000): *Computer-Aided Reasoning: Case Studies*. Kluwer Academic Publishers, doi:10.1007/978-1-4757-3188-0.

[24] Matt Kaufmann & J Strother Moore (2025): *ACL2 homepage*. Available at `https://www.cs.utexas.edu/users/moore/acl2/`.

[25] Scott Kausler & Elena Sherman (2014): *Evaluation of string constraint solvers in the context of symbolic execution*. In Ivica Crnkovic, Marsha Chechik & Paul Grünbacher, editors: *ACM/IEEE International Conference on Automated Software Engineering, ASE '14*, ACM, pp. 259–270, doi:10.1145/2642937.2643003.

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

[27] Ankit Kumar, Andrew T. Walter & Panagiotis Manolios (2022): *Automated Grading of Automata with ACL2s*. In Pedro Quaresma, João Marcos & Walther Neuper, editors: *Proceedings 11th International Workshop on Theorem Proving Components for Educational Software, ThEdu@FLoC 2022*, *EPTCS* 375, pp. 77–91, doi:10.4204/EPTCS.375.7.

[28] Panagiotis Manolios & Sudarshan K. Srinivasan (2004): *A Suite of Hard ACL2 Theorems Arising in Refinement-Based Processor Verification*. In: *Proceedings of the Fifth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 '04)*.

[29] Panagiotis Manolios & Sudarshan K. Srinivasan (2005): *Verification of executable pipelined machines with bit-level interfaces*. In: *2005 International Conference on Computer-Aided Design, ICCAD 2005*, IEEE Computer Society, pp. 855–862, doi:10.1109/ICCAD.2005.1560182.

[30] Panagiotis Manolios & Sudarshan K. Srinivasan (2006): *A Framework for Verifying Bit-Level Pipelined Machines Based on Automated Deduction and Decision Procedures*. *J. Autom. Reason.* 37(1-2), pp. 93–116, doi:10.1007/S10817-006-9035-0.

[31] 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.

[32] 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.

[33] 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.

[34] Panagiotis Manolios & Daron Vroon (2006): *Termination Analysis with Calling Context Graphs*. In Thomas Ball & Robert B. Jones, editors: *Computer Aided Verification*, *Lecture Notes in Computer Science* 4144, Springer, pp. 401–414, doi:10.1007/11817963_36.

[35] Leonardo Mendonça de Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In C. R. Ramakrishnan & Jakob Rehof, editors: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Proceedings*, *LNCS* 4963, Springer, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[36] Yan Peng & Mark R. Greenstreet (2015): *Extending ACL2 with SMT Solvers*. In Matt Kaufmann & David L. Rager, editors: *Proceedings Thirteenth International Workshop on the ACL2 Theorem Prover and Its Applications*, *EPTCS* 192, pp. 61–77, doi:10.4204/EPTCS.192.6.

[37] Yan Peng & Mark R. Greenstreet (2018): *Smtlink 2.0*. In Shilpi Goel & Matt Kaufmann, editors: *Proceedings of the 15th International Workshop on the ACL2 Theorem Prover and Its Applications*, *EPTCS* 280, pp. 143–160, doi:10.4204/EPTCS.280.11.

[38] Justin Slepak, Panagiotis Manolios & Olin Shivers (2018): *Rank polymorphism viewed as a constraint problem*. In Sven-Bodo Scholz & Olin Shivers, editors: *Proceedings of the 5th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming, ARRAY@PLDI 2018*, ACM, pp. 34–41, doi:10.1145/3219753.3219758.

[39] Sudarshan Kumar Srinivasan (2007): *Efficient verification of bit-level pipelined machines using refinement*. Ph.D. thesis, USA. AAI3294559.

[40] Andrew T. Walter: *Lisp-Z3 Repo*. `https://github.com/mister-walter/cl-z3`.

[41] Andrew T. Walter, Benjamin Boskin, Seth Cooper & Panagiotis Manolios (2019): *Gamification of Loop-Invariant Discovery from Code*. In Edith Law & Jennifer Wortman Vaughan, editors: *Proceedings of the Seventh AAAI Conference on Human Computation and Crowdsourcing, HCOMP 2019*, AAAI Press, pp. 188–196, doi:10.1609/HCOMP.V7I1.5277.

[42] Andrew T. Walter, David A. Greve & Panagiotis Manolios (2022): *Enumerative Data Types with Constraints*. In Alberto Griggio & Neha Rungta, editors: *22nd Formal Methods in Computer-Aided Design, FMCAD 2022*, IEEE, pp. 189–198, doi:10.34727/2022/ISBN.978-3-85448-053-2_25.

[43] Andrew T. Walter, Ankit Kumar & Panagiotis Manolios (2023): *Proving Calculational Proofs Correct*. In Alessandro Coglio & Sol Swords, editors: *Proceedings of the 18th International Workshop on the ACL2 Theorem Prover and Its Applications*, *EPTCS* 393, pp. 133–150, doi:10.4204/EPTCS.393.11.

[44] 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.

An Enumerative Embedding of the Python Type System in ACL2s

Samuel Xifaras Panagiotis Manolios Andrew T. Walter William Robertson

Khoury College
Northeastern University
Boston, Massachusetts, USA
{xifaras.s,p.manolios,walter.a,w.robertson}@northeastern.edu

Python is a high-level interpreted language that has become an industry standard in a wide variety of applications. In this paper, we take a first step towards using ACL2s to reason about Python code by developing an embedding of a subset of the Python type system in ACL2s. The subset of Python types we support includes many of the most commonly used type annotations as well as user-defined types comprised of supported types. We provide ACL2s definitions of these types, as well as `defdata` enumerators that are customized to provide code coverage and identify errors in Python programs. Using the ACL2s embedding, we can generate instances of types that can then be used as inputs to fuzz Python programs, which allows us to identify bugs in Python code that are not detected by state-of-the-art Python type checkers. We evaluate our work against four open-source repositories, extracting their type information and generating inputs for fuzzing functions with type signatures that are in the supported subset of Python types. Note that we only use the type signatures of functions to generate inputs and treat the bodies of functions as black boxes. We measure code coverage, which ranges from about 68% to more than 80%, and identify code patterns that hinder coverage such as complex branch conditions and external file system dependencies. We conclude with a discussion of the results and recommendations for future work.

1 Introduction

Python is an industry-standard language that is used in software engineering disciplines ranging from web development to machine learning [14]. Its versatility and ease of use have propelled it to its position as the second most popular programming language by usage on GitHub, behind only JavaScript [3]. With this popularity has come many proposed improvements to the language, including type annotations which were introduced in Python Enhancement Proposal (PEP) 484 [41]. This is a reasonable addition, as static typing in programming has been shown to have a host of benefits [17]. As these type annotations continue to be adopted by developers, they represent a rich source of data for application analysis.

Since Python software has become ubiquitous, this paper is motivated by the need for robust software verification in Python. Python code is typically tested with unit testing, which tests a property of the code in a single example scenario. Although unit testing can be effective when done well, it has a reputation for being burdensome, and there is evidence that developers find it challenging–or are not motivated– to cover deeply nested code [53]. In this paper, we introduce an *extensible*, *enumerative* embedding of the Python type system that can generate representative examples of simple and complex Python types, and evaluate its effectiveness by measuring code coverage on real-world open source repositories. The representative examples generated by this embedding can drive fuzzing and property-based testing (PBT), among other techniques.

Importantly, the goal of this work is not to model the semantics of the type system. This is mostly because it is unclear what the semantics of the type system of Python are. Rak-amnouykit et al. [38], for

Gamboa and Manolios (Eds):
ACL2 Workshop 2025
EPTCS 423, 2025, pp. 124–144, doi:10.4204/EPTCS.423.11

instance, find that two of the most popular Python type checkers, *mypy* [46] and *pytype* [2], implement and check different type systems. Nor is it the goal to model the semantics of Python, which are complex and subject to change in new Python versions. For a preview of the complex considerations involved in modeling the semantics of Python, including scoping and generators, see [35]. Instead, our goal is to facilitate fuzzing and PBT of Python programs with an embedding of the Python type system that supports extension and can be used to generate test cases.

Our embedding is *extensible* in that it can be extended with new types. This is necessary as Python primitive types can be composed in infinitely many ways with compound types such as lists, tuples, and dictionaries. User-defined classes are also frequently used by Python programmers, and we support embedding classes as record types, as long as their field types are recursively representable in the embedding. The embedding also supports extension with union types. How the type system works and how it can be extended are discussed in Section 4.

Our embedding is *enumerative* because it leverages the enumerative data definition framework of ACL2s [12]. We say a data type is enumerative when it is associated with an enumerator function that maps natural numbers to elements of the type. After the embedding is extended to include user-defined types, the enumerative property of the embedding allows examples of these types to be generated immediately. This is useful for test data generation in unit testing, producing large numbers of test cases for PBT, and producing seed inputs for fuzzing. We discuss this aspect of the embedding further in Section 5.

We chose to implement this embedding in ACL2s because we view it as uniquely positioned to support the goals of this work. Its logical foundation enables formal reasoning and theorem proving for Python types and constructs, and the ACL2s `defdata` framework [12] grants the enumerative property of the embedding. Taken together, these enable seamless integration with dynamic approaches, such as fuzzing and PBT. Accordingly, the contributions of this paper are being used in ongoing work related to fuzzing in Python.

In summary, we claim the following contributions in this paper:

- **An extensible, enumerative embedding** of the Python type system in ACL2s. The code is open-sourced and available.[1]
- **Custom enumerators** for Python primitive types. The enumerators are designed to produce representative examples of Python types.
- **An evaluation** of code covered in four open source repositories with inputs generated by the embedding's enumerators.

Note that conversion of the embedding's generated examples to Python objects is not specified in this paper. Details are given in Xifaras's master's thesis [52], and we also plan to publish a specification of the conversion process in a forthcoming software engineering paper. To motivate why this problem is interesting and nontrivial in the context of our ongoing work in fuzzing, consider the fact that pickle [1], Python's built-in object serialization protocol, does not flatly encode the data. It instead encodes objects as instructions to a virtual machine that, when played back, reconstruct the original object [45]. We have observed that this storage format is far from ideal for fuzzing and poses significant security concerns [52].

We invite the community to extend the embedding and implement new features or program analyses on top of it. There are many features of the type system that are missing from our embedding, such as protocol types [28], and new features are added with every update to the Python language. This leaves many opportunities for future work.

[1] https://github.com/acl2/acl2/tree/master/books/projects/python/embedding

The remainder of the paper is organized as follows. In Section 2, we discuss some background information and a motivating example. Section 3 gives an overview of related work. Section 4 describes how the model is embedded and how it can be extended. Section 5 contains definitions of custom enumerators that produce representative examples of Python primitive types. We provide examples of the usage of the embedding in Section 6, and we evaluate its performance on several open source repositories in Section 7. We discuss the results and future work in Section 8, and conclude in Section 9.

2 Background and Motivating Example

As Python has seen increasing use across the software engineering industry, the need to collaborate in large-scale codebases has grown. This has led to the emergence of static typing in Python through the PEP-484 system of type hints [41]. These hints are optional and not checked by the language implementation. *Type checkers*, such as mypy [46] and pytype [2], have emerged to statically verify the correctness of these type hints. In fact, Python's type system semantics are heavily influenced by mypy's design decisions [26, 41]. Due, however, to the complex semantics of Python and the complexity of the type hints themselves [43], these type checkers are neither sound nor complete [38]. When Python code is deployed in a production environment, false negatives are particularly dangerous because they could lead to unhandled crashes, resulting in reduced availability, customer dissatisfaction, and developer frustration.

Consider, for instance, Listing 1. This listing contains a function, `create_decimal`, that takes an integer and a floating-point number, converts them to their string representations, concatenates them, then converts the result to a float and returns it. This is a seemingly innocuous function. Its stated purpose is to construct a floating-point value out of an integer component representing the digits to the left of a decimal point and a floating-point value representing the digits to the right of a decimal point. At the time of this writing, the latest versions of two Python type checkers on Python 3.12, mypy (v1.14.1) and pytype (v2024.10.11), do not report any errors in this definition. However, there are several corner cases that this function does not account for that trigger unhandled exceptions. For instance, consider when the floating-point argument takes a negative value, or a value of `inf` (infinity), or `nan` (not-a-number). When an integer is concatenated with the string `"nan"`, the result cannot be cast back into a float, resulting in an exception. Property-based testing with input data that covers these special-case floating-point values can find this issue, highlighting the importance of both strong testing discipline and representative test data.

```
1 def create_decimal(whole_part: int, decimal_part: float) -> float:
2     """Create a decimal number from the given whole part and decimal part"""
3     return float(str(whole_part) + str(decimal_part).lstrip('0'))
```

Listing 1: Motivating example: A function that behaves well when `decimal_part` is non-negative and not `"nan"`, `"inf"`, or `"-inf"`.

The embedding we present in this paper enables the discovery of this and other bugs in Python programs through the generation of representative test data. In the case of the function given in Listing 1, we find this bug because of the customized enumerative floating-point data type that accurately represents Python's `float` type. Our goal with this work is to augment type checking with dynamic execution and formal reasoning, thereby enabling Python developers to extract more value from their tooling and investment in type annotations. We hope this grants developers greater confidence in their code.

3 Related Work

In this section, we cover various areas of the literature that are related to our work, and offer brief discussions about how our work fits in with each.

Type annotations in Python. The Python language has gone through many iterations, in particular with its system of type annotations (also referred to as type hints). Multiple PEP documents have been published about Python's type system, such as PEP-484 [41], the specification of type hints in Python, PEP-483 [42], which specifies the theory behind type hints, and PEP-544 [28] which introduces structural subtyping into the language. Di Grazia and Pradel [16] perform a comprehensive study of open source Python code and measure the state of type annotation usage in the ecosystem. They find that there is an upward trend in type annotation usage, but fewer than 10% of code elements are annotated. There are also distinct usage patterns among different repositories, and they find that repositories with higher numbers of contributors tend to utilize type annotations more. The upward trend in usage of Python type annotations is a positive signal for the potential adoption of our work.

Type checking in Python. We have chosen Python types as an aspect of Python to model in ACL2s because they are relatively simple to model, we anticipate that type annotations' popularity will continue to grow, and tooling for type checking in Python has several issues. Third-party tooling for supporting the Python type system is necessary because PEP-484 clearly states that it is not the intent of the Python implementation to statically check types [41]. Static type checkers have therefore arisen to offer compile-time guarantees of type safety. *mypy* [46] and *pytype* [2] are notable examples. Rak-amnouykit et al. [38] perform a study on the outputs of these type checkers, and they find that mypy and pytype implement two different type systems. They also find, in corroboration of the results of Di Grazia and Pradel [16], that statically detectable type errors often do not seem to inhibit developers from committing code. This suggests that although type annotations are seeing increased use, there is much work to do in fixing type errors in Python code. Evidence that this is a practical problem is provided by Rak-amnouykit et al. who find that these tools emit false positives [38]. Finally, Roth shows that Python type hints are Turing complete [43], indicating that type checking in Python is an undecidable problem. No solution can be both sound and complete.

Formal verification of Python. Given Python's importance in the modern software landscape, formal verification of its semantics is an appealing academic pursuit. While the goal of the present work is not to formally verify Python programs, we consider these works related because they relate to the theme of tooling for Python software verification. Several attempts have been made to formally specify or verify subsets of Python. Ranson formally specifies an operational semantics of a heavily restricted subset of Python 2 called *IntegerPython*, which only has integers and booleans as data types [39]. *IntegerPython* is implemented in the Isabelle/HOL proof assistant, and Ranson proves correctness of a Turing machine simulator written in the language [39]. Politz et al. implement a small-step operational semantics for Python, and they contribute a translator from general Python programs to a "core language" for which the semantics are modeled [35]. They test their implementation on many test cases from the CPython implementation. Smeding, in a master's thesis regarded by Politz et al. as "sadly unheralded" [35], implements an executable semantics in literate Haskell and tests it against 134 test cases [44]. Smeding's semantics are for Python 2.5, however, which is a deprecated version of the language. Also in a master's thesis, Köhl implements an operational, executable semantics of Python using the $\mathbb{K}$ semantic framework [24]. Köhl's semantics are based on Python 3.7, which is also a deprecated version of the language, but closer to the language's current state. To our knowledge, there are no works that formally specify or verify any part of Python in ACL2/ACL2s, rendering the present work the first to do so.

Fuzzing in Python. As previously mentioned, we are engaged in ongoing work on utilizing the

```
(defun add-nonparametric-type (name defdata-ty)
  (setf (gethash name *type-table*)
        `(:name ,name
          :kind "nonparametric"
          :defdata-ty ,defdata-ty)))

(defun add-parametric-type (name defdata-ty-lambda)
  (setf (gethash name *type-table*)
        `(:name ,name
          :kind "parametric"
          :defdata-ty ,defdata-ty-lambda)))

(defun add-alias-type (name alias-of)
  (let ((name (string-downcase name))
        (alias-of (string-downcase alias-of)))
    (when (equal (gethash alias-of *alias-table*) name)
          (error "It is illegal to set ~a as an alias for ~a because ~a is already an alias for ~a
              ." name alias-of alias-of name))
    (setf (gethash name *alias-table*)
          alias-of)))
```

Listing 2: Definitions of `add-nonparametric-type`, `add-parametric-type`, and `add-alias-type`.

enumerative embedding for fuzzing Python code. Fuzzing in Python is a nascent area of study as Python becomes increasingly widespread in industry. PyRTFuzz [29] is a recent paper that proposes an approach to fuzzing the Python interpreter, and claims several bug discoveries. Our ongoing work focuses on fuzzing arbitrary Python code, rather than the interpreter, and the ACL2s-based type example generation introduced in the present paper is used to create seed inputs for fuzzing. HypoFuzz [19], maintained by Zac Hatfield-Dodds, is a Python library based on the PBT library Hypothesis [30] that uses advanced fuzzing techniques and long time budgets to find counterexamples to properties. In his master's thesis, Xifaras covers the embedding presented here in greater depth, as well as how it integrates with a larger fuzzing system [52]. Experimental results on fuzzing in Python are also presented [52].

ACL2s. The ACL2 Sedan (ACL2s) [15, 9] is an extension of the ACL2 theorem prover[20, 21, 22]. On top of the capabilities of ACL2, ACL2s provides the following: 1) a powerful type system via the `defdata` data definition framework [12] and the `definec` and `property` forms, which support typed definitions and properties, 2) counterexample generation capability via the `cgen` framework, which is based on the synergistic integration of theorem proving, type reasoning and testing [11, 13, 10], 3) a powerful termination analysis based on calling-context graphs [34] and ordinals [31, 32, 33], 4) an (optional) Eclipse IDE plugin [9], and 5) the ACL2s systems programming framework (ASPF) [50] which enables the development of tools in Common Lisp that use ACL2, ACL2s and Z3 as a service [51, 48, 47, 25, 49]. Our work builds on ACL2s and uses its data definition framework to model Python types. Walter et al. also build on the enumerative data types in ACL2s, adding dependent types and showing how dependent type enumerators can generate a great breadth of examples of 802.11 Wi-Fi packets [48]. These "enumerative data types with constraints" may become useful for this embedding in future work. We additionally leverage the aforementioned ACL2s systems programming framework in our work to enable integration with foreign function interfaces and other application libraries such as an HTTP server.

```
1  (defun init-types ()
2    (add-nonparametric-type "integer" 'acl2s::py-integer)
3    (add-alias-type "int" "integer")
4    (add-nonparametric-type "float" 'acl2s::py-float)
5    (add-nonparametric-type "bool" 'acl2s::py-bool)
6    (add-nonparametric-type "unicode-codepoint-string"
7                            'acl2s::unicode-codepoint-string)
8    (add-alias-type "unicode" "unicode-codepoint-string")
9    (add-alias-type "str" "unicode-codepoint-string")
10   (add-alias-type "boolean" "bool")
11   (add-parametric-type "list"
12    (lambda (el-ty)
13      (let ((elt-ty-sym (translate-type-to-defdata
14                         (if (stringp el-ty) el-ty (alist-to-plist el-ty)))))
15        `(acl2s::listof ,elt-ty-sym))))
16   (add-parametric-type "dictionary"
17    (lambda (key-ty val-ty)
18      (let ((key-ty-sym (translate-type-to-defdata
19                         (if (stringp key-ty) key-ty (alist-to-plist key-ty))))
20            (val-ty-sym (translate-type-to-defdata
21                         (if (stringp val-ty) val-ty (alist-to-plist val-ty)))))
22        `(acl2s::map ,key-ty-sym ,val-ty-sym))))
23   (add-parametric-type "fixedtuple"
24    (lambda (&rest types)
25      (let ((ty-syms (mapcar (lambda (ty)
26                               (translate-type-to-defdata
27                                 (if (stringp ty) ty (alist-to-plist ty))))
28                             types)))
29        `(acl2s::list ,@ty-syms))))
30   (add-nonparametric-type "nonetype" 'acl2s::py-none)
31   (add-nonparametric-type "bytes" 'acl2s::py-bytes))
```

Listing 3: Initial setup of the type table.

4 Embedding Construction

To implement the embedding, we leverage ACL2s Systems Programming [50] to create an API in Common Lisp that makes calls which affect an underlying ACL2s theory (the "world"). We use ACL2s Systems Programming as a foundation because it simplifies interfacing with foreign systems. In our ongoing fuzzing work, for instance, we implement an HTTP server on top of the Common Lisp API that accepts requests to update the embedding with new types and get examples of embedded types. Xifaras describes this HTTP interface in [52]. At this stage of the implementation, the only ACL2s calls that are being made are `defdata` calls, which in turn make several calls to `defthm`.

The embedding uses two data structures to track information about known types: a *type table* and an *alias table*. The type table is a hash table that maps type names to property lists (*plists*) that contain metadata about the types. Lines 1-11 of Listing 2 show the two functions that extend this table, `add-nonparametric-type` and `add-parametric-type`.

As shown in Listing 2, the values of the hash table are plists that have three keys, `:name`, `:kind`, and `:defdata-ty`. If the type's kind is non-parametric, `defdata-ty` takes the value of an S-expression containing the `defdata` definition syntax. See [12] for a reference on this syntax. Otherwise, it takes

a lambda which defines how to produce the `defdata` definition expression from the parameters of the type.

Types may be known by different names, or a programmer may want to assign multiple names to the same underlying type. Type aliases enable this. Extension of the type alias table is done via the `add-alias-type` function, whose definition is also given in Listing 2 (lines 13-19). This function takes two string values that represent the alias and the name to be aliased. They are both converted to lowercase (lines 14-15) to maintain case insensitivity. Lines 16-17 perform a simple cycle check, to ensure that the name to be aliased is not already an alias for the given alias. This check could be generalized to arbitrary-length cycles.

The model of the type system starts with a set of base types defined in Common Lisp as shown in Listing 3. Note the use of `add-nonparametric-type`, `add-parametric-type`, and `add-alias-type` as defined in Listing 2. This set of initial types was mostly derived from the set of most commonly used types in annotations, as identified by the work of Di Grazia and Pradel [16]. The `acl2s` symbols that are shown are associated with `defdata` definitions of the embedding.

The embedding can also be extended with complex types in Python. Any user-defined class that has field types that are recursively representable in the embedding can be admitted to the model, with the caveat that the embedding does not support self-referential or mutually recursive class definitions at this time. Admission of recursively representable types is implemented by an iterative type extraction procedure that continues until a maximum number of iterations has been reached or until a fixed point.

This extraction process is given in Algorithm 1. Lines 1-4 set up the state variables. S' is the final set of types, S'_{prev} maintains the set from the previous iteration to check whether a fixed point has been reached. On line 3, C is set to the domain of A, which is the set of extracted user-defined classes in the subject codebase. In this definition, *maxIters*, the maximum number of iterations to perform if no fixed point is reached, is set to 5. If the time budget allows, it is advisable to increase this value so that as many user-defined types can be registered as possible. The remainder of the lines in this algorithm define the main loop. Line 7 contains the check for whether a type can be registered under the current model. *types* is a helper function that returns the types used in a function signature. `registerType` (line 8) registers (i.e. admits) the type in the embedding. Refer to Xifaras for further details on this extraction [52].

Listing 4 contains examples of user-defined types that can be registered with the type extraction process. The second of the two classes, `TestClassB`, references the first, `TestClassA`. These class types are used in the definitions of two functions, `use_a` and `use_b`. The `defdata` calls used to embed this type information are given in Listing 5. Note that the listed `defdata` calls fully represent all types that appear in this example. This implies that `use_a` and `use_b` are extractable as "appropriate functions," as defined in Section 7.

4.1 Note on Embedding Philosophy

Note that we have chosen to adopt a conservative philosophy when embedding classes in Python. The runtime type system of Python has duck typing semantics, and Python supports runtime operations that add and remove arbitrary attributes of objects while preserving the Python `isinstance` relation, which checks that an object is an instance of a given class. This means that, at runtime, an instance of class `Bar` that behaves exactly like an instance of class `Foo` (if it walks like a duck, talks like a duck...) can be passed off as an instance of `Foo`, and an instance of `Foo` can be mutated to look like an instance of `Baz`, at which point code that operates on instances of `Foo` no longer recognize it as an object of type `Foo`, except by `isinstance`. Attempting to generate examples that capture the broadness of duck typing semantics may be "representative" of what is possible in Python, but this may lead to error reports that would be

Algorithm 1 Type Registration

Input: Initial type set S; mapping of class name to set of attribute types A; mapping of class name to set of method signatures M
Output: Final type set S'

$S' \leftarrow S$
$S'_{prev} \leftarrow S'$
$C \leftarrow \text{dom}(A)$
$maxIters \leftarrow 5$
for $i \leftarrow 1$ to $maxIters$ **do**
    for each $c \in C$ **do**
        if $A(c) \subseteq S'$ **and** $\left(\bigcup_{m \in M(c)} \text{types}(m)\right) \subseteq S'$ **then**
            registerType(c)
            $S' \leftarrow S' \cup \{c\}$
        end if
    end for
    if $S' = S'_{prev}$ **then** ▷ check for fixed point
        break
    end if
    $S'_{prev} \leftarrow S'$
end for
return S'

easily rejected by a user as false positives, since an instance of `Foo` that looks and acts like an instance of `Baz`, for instance, would never be created by their code. This is what we mean by "conservative." We take the type annotations and corresponding class definitions at face value, in the same way that mypy does [27].

5 Custom Enumerators

In ACL2s, data types are *enumerative*. This means that each type is associated with an enumerator function that maps the natural numbers to examples of the type [12]. This is useful in the context of fuzzing and property-based testing because if one can define a type of data in ACL2s, one immediately has access to examples of it. We say that each data type in ACL2s has an enumerator *attached* to it, and the attached enumerator can be changed programmatically.

In our ongoing work on fuzzing in Python, we found that the default ACL2s `defdata` enumerators for certain primitive types do not produce a wide variety of examples. These default enumerators are intended to produce examples that would be readable by a student in case one causes their code to fail [13]. We are not concerned with readability of the examples, so we instead defined custom enumerators that produce a much wider range of values suitable for fuzzing. The definitions of these custom enumerators are given in this section.

5.1 Integers

To test code that works with integers, we have created a custom enumerator for integers that generates small-magnitude and very large-magnitude integer values of positive and negative sign. The Python integer type has arbitrary precision, but Python code often interfaces with native code which uses machine integers that may be 8, 16, 32, or 64 bits. Feedback from running code annotated with Python integers

```
1  from typing import List, Tuple
2
3  class TestClassA:
4      def __init__(self, a: float, b: List[int]) -> None:
5          self.a = a
6          self.b = b
7
8  class TestClassB:
9      def __init__(self, a: int, b: TestClassA) -> None:
10         self.a = a
11         self.b = b
12
13 def use_a(a: TestClassA) -> Tuple[float, List[int]]:
14     return (a.a, a.b)
15
16 def use_b(b: TestClassB) -> Tuple[int, TestClassA]:
17     return (b.a, b.b)
18
19 a_inst = TestClassA(3.5, [1, 2, 3])
20 b_inst = TestClassB(4, a_inst)
21
22 use_a(a_inst)
23 use_b(b_inst)
```

Listing 4: Example of Python class definitions and functions that use them.

can help the programmer narrow down the integer type that their code actually expects. For example, a function that makes a call to a native routine in the popular library *numpy* [18] may be annotated as taking a Python integer, but any integer that does not fit within 64 bits may cause unexpected behavior because the underlying native code expects a `numpy.int64` value. The custom enumerator generates integers from several cases with probabilities given in Table 1. For convenience of notation, where $l,i,h \in \mathbb{Z}$, let $P_2^+(l,h) := \{2^i \mid l \leq i \leq h\}$, $P_2^-(l,h) := \{-2^i \mid l \leq i \leq h\}$, and $P_2^\pm(l,h) := P_2^+(l,h) \cup P_2^-(l,h)$.

Description	**%**	**Set**
Sum of powers of two	85	$\{a+b \mid a \in P_2^\pm(0,64) \wedge b \in P_2^\pm(0,16)\}$
65-bit integers	6	$UnionAll(\{\{a,-a\} \mid 2 \leq a \leq 2^{65}\})$
Powers of 2, with off by one	6	$UnionAll(\{\{a,a-1,a+1\} \mid a \in P_2^\pm(0,65)\})$
One	1	$\{1\}$
Zero	1	$\{0\}$
Negative one	1	$\{-1\}$

Table 1: Custom integer enumerator cases

As an example of how one may define a custom enumerator the integer enumerator source code is given in Listing 6. Note the correspondence between this definition and Table 1. The probability distribution is given on line 5, in the expression `'(85 6 6 1 1 1)`. Lines 6-21 define the cases of the

```
1  (DEFDATA TY1039 PY-INTEGER)  ;; int
2  (DEFDATA TY1041 PY-FLOAT)  ;; float
3  (DEFDATA TY1043 (LISTOF TY1039) DO-NOT-ALIAS T)  ;; List[int]
4
5  ;; class TestClassA[a: float, b: List[int]]
6  (DEFDATA ACL2S::CLASSTEST.TESTCLASSA
7           (DEFDATA::RECORD (A . ACL2S::TY1041) (B . ACL2S::TY1043))
8
9  ;; class TestClassB[a: int, b: TestClassA]
10 (DEFDATA ACL2S::CLASSTEST.TESTCLASSB
11          (DEFDATA::RECORD (A . ACL2S::TY1039)
12           (B . ACL2S::CLASSTEST.TESTCLASSA))
13
14 ;; Tuple[float, List[int]]
15 (DEFDATA TY1096 (LIST TY1041 TY1043))
16
17 ;; Tuple[int, TestClassB]
18 (DEFDATA TY1100 (LIST TY1039 CLASSTEST.TESTCLASSB))
```

Listing 5: `defdata` calls issued by type extraction procedure when analyzing Listing 4.

enumerator. In the first case, two values are generated then added. In the second case, three 32 bit integers are generated, and then a helper function, `make-nat-upto-2-expt-65`, is called to produce a 65-bit integer. In the third case, a power of two is generated, and an offset of either -1, 0, or 1 is selected by generating a random number between zero and two and subtracting one from it (lines 17-18). The remaining three cases are trivial.

5.2 Strings

Python supports Unicode strings, so our enumerator generates many different varieties of Unicode strings. The probabilities are broken down as specified in Table 2. Note the use of $C_n(S)$ notation. We let the notation $C_n(S)$ denote the set of strings of length at most n composed of characters in S. Formally, $C_n(S) := \{c_0 \dots c_i \dots c_n \mid c_0 \in S \wedge \dots \wedge c_i \in S \wedge \dots \wedge c_n \in S\}$. The set ASCII denotes the set of ASCII characters. The Emoji set denotes the set of Emoji Unicode characters. The set Gr denotes the set of Greek letter characters. The set MathSym denotes the set of mathematical symbol characters. The set LtnDiac denotes the set of latin letters with diacritic marks (such as ä and á). The set CmpEmoji denotes the set of compound emojis, which are emoji characters that span two or more codepoints.

Note that there is a distinction between string *literals* and the `str` class in Python. There are several string literals in Python that have different semantics. The standard string literal, denoted with quotes (`""`), produces an instance of the `str` class. The `str` class, according to Python's documentation, represents a sequence of Unicode codepoints [36]. Our enumerator produces sequences of Unicode codepoints, satisfying this definition. Raw strings, denoted `r""`, also produce `str` instances, but escape sequences are ignored. "F-strings," short for format strings, are denoted `f""`. These string literals support `printf`-like interpolation of variables into strings, and they also produce `str` instances. Byte string literals, despite having similar syntax to the aforementioned literals (`b""`), produce `bytes` instances. The `bytes` type represents a sequence of 8-bit integers, and its representation in the embedding is defined in the `defdata` framework as shown in Listing 7.

```
(defun python-int-enum/acc (n seed)
  (declare (xargs :mode :program))
  (declare (ignore n) (type (unsigned-byte 31) seed))
  (b* (((mv choice seed)
        (defdata::weighted-switch-nat '(85 6 6 1 1 1) seed)))
    (case choice
      (0 (b* (((mv val-64 seed) (signed-power-of-two-enum-seed 0 64 seed))
              ((mv val-16 seed) (signed-power-of-two-enum-seed 0 16 seed)))
           (mv (+ val-64 val-16) seed)))
      (1 (b* (((mv r1 seed) (defdata::genrandom-seed (1- (expt 2 31)) seed))
              ((mv r2 seed) (defdata::genrandom-seed (1- (expt 2 31)) seed))
              ((mv r3 seed) (defdata::genrandom-seed (1- (expt 2 31)) seed))
              (v (make-nat-upto-2-expt-65 r1 r2 r3))
              ((mv sign seed) (random-bool seed)))
           (mv (* (if sign 1 -1) (1+ v)) seed)))
      (2 (b* (((mv pow-2 seed) (signed-power-of-two-enum-seed 1 65 seed))
              ((mv constant+1 seed) (switch-nat-safe-seed 3 seed)))
           (mv (+ pow-2 (1- constant+1)) seed)))
      (3 (mv -1 seed))
      (4 (mv 0 seed))
      (t (mv 1 seed)))))

(defun python-int-enum (n)
  (declare (xargs :mode :program))
  (b* (((mv x &) (python-int-enum/acc 0 n)))
    x))
```

Listing 6: The integer enumerator.

```
(defnatrange u8 (expt 2 8)) ;; alternatively, (defdata u8 (range integer (0 <= _ < (expt 2 8))))
(defdata py-bytes (listof u8) :do-not-alias t)
```

Listing 7: The `defdata` of Python's `bytes` type.

5.3 Floats

Python, like many other programming languages, has a `float` type, which represents a 64-bit double-precision IEEE floating-point number. Although ACL2s does not have a built-in `float` type, it supports arbitrary precision rational numbers. We observe that all values a floating-point number can take are rational numbers, except for the special values of `-inf`, `inf`, and `nan`. Therefore, we represent the Python floating-point type as a union between the ACL2s rationals and ACL2s representations of the special floating-point values.

In order to define a floating-point type that is representative of Python's float type, we define a custom enumerator that produces rational numbers in predefined interesting categories, as well as the aforementioned special case values. We use the same $P_2^+/P_2^-/P_2^\pm$ notation from the previous integer enumerator definition. There is bias towards generating "edge case" values that may be likely to trigger interesting behavior. Table 3 contains the cases of the enumerator. Note that the terms "normal" and "subnormal" are used. A "normal" floating-point number is one that has no leading zeros in its mantissa. A "subnormal" floating-point number is one that has one or more leading zeros in its mantissa.

Description	%	Set
ASCII strings	50	$C_{10^4}(\text{ASCII})$
Emoji strings	2	$C_{10^4}(\text{Emoji})$
Greek-letter strings	2	$C_{10^4}(\text{Gr})$
Mathematical symbols	2	$C_{10^4}(\text{MathSym})$
Latin diacritics	2	$C_{10^4}(\text{LtnDiac})$
Compound emojis	2	$C_{10^4}(\text{CmpEmoji})$
Mixed strings	40	$C_{10^4}(\text{ASCII} \cup \text{Emoji} \cup \text{Gr} \cup \text{MathSym} \cup \text{LtnDiac} \cup \text{CmpEmoji})$

Table 2: Custom string enumerator cases.

6 Usage

The functionality of this embedding is exposed through a Common Lisp API. In this section, we cover usage examples of several of these API calls. Available functions include a setter for the random seed, the `add-parametric-type` and `add-nonparametric-type` functions for extending the type system, and the `generate-examples` function for retrieving examples.

```
1  (include-book "top")                  ;; load the embedding's ACL2s book
2  :q                                    ;; quit into raw lisp
3  (load "api.lsp")                      ;; Load the backend module, which contains the API
4  (in-package :acl2s-python-types)      ;; "acl2s-python-types" is the name of the API package
5
6  (defvar *enum-random-state*)               ;; create variable to hold random state
7  (setf *enum-random-state* (make-cl-seed 1)) ;; Set seed
8
9  ;; Generate 100 examples of floats
10 (generate-examples "float" 100 *enum-random-state*)
11
12 ;; Register a union between integers, floats, and strings called intfloatstr
13 (register-union "intfloatstr" '("int" "float" "str"))
14
15 ;; Generate 100 examples of intfloatstr
16 (generate-examples "intfloatstr" 100 *enum-random-state*)
```

Listing 8: Example of API usage.

Before any API calls can be used, the environment must be initialized properly. Lines 1-7 of Listing 8 contain setup code that is needed to import the ACL2s book, exit into the Common Lisp environment, load the Common Lisp API module (`api.lsp`), then initialize the random state. In the code listing, the working directory is assumed to be the root of the embedding implementation's source code.

Now that the environment is initialized, examples of types can be generated. Line 10 of Listing 8 contains a call to `generate-examples` that generates 100 examples of Python floating-point numbers. If you were to run this code, it would produce an S-expression containing rational numbers and occasionally

Description	%	Set
Rational numbers	76	$\left\{ n/k \mid n,k \text{ generated using the integer enumerator} \right\}$
Powers of 2 with small-magnitude exponents	5	$UnionAll\left(\{\{a, a-1, a+1\} \mid a \in P_2^{\pm}(-64, 64)\}\right)$
Powers of 2 with large-magnitude exponents	5	$UnionAll\left(\begin{array}{r}\{\{a, a-1, a+1\} \mid a \in P_2^{\pm}(65, 1024) \cup \\ P_2^{\pm}(-1024, -65)\}\end{array}\right)$
Min and max normal 32-bit floats	3	$\left\{\begin{array}{l} 2^{-126}, 2^{-126}+1, 2^{-126}-1, \\ 2^{127}(2^{-23}-2), 2^{127}(2^{-23}-2)-1, \\ 2^{127}(2^{-23}-2)+1 \end{array}\right\}$
Min and max normal 64-bit floats, with off by one	3	$\left\{\begin{array}{l} 2^{-1022}, 2^{-1022}-1, 2^{-1022}+1, \\ 2^{1023}(2^{-52}-2), 2^{1023}(2^{-52}-2)-1, \\ 2^{1023}(2^{-52}-2)+1 \end{array}\right\}$
Max integer representable as a 32 or 64-bit float	2	$\{2^{24}, -2^{24}, 2^{53}, -2^{53}\}$
Min and max subnormal 32 and 64-bit floats	2	$\left\{\begin{array}{l} 2^{-149}, -2^{-149}, 2^{-126}(1-2^{-23}), \\ -2^{-126}(1-2^{-23}), 2^{-1074}, -2^{-1074}, \\ 2^{-1022}(1-2^{-52}), -2^{-1022}(1-2^{-52}) \end{array}\right\}$
Not-a-number	1	{`nan`}
Positive infinity	1	{`inf`}
Negative infinity	1	{`-inf`}
Negative zero	1	$\{-0\}$

Table 3: Custom floating-point number enumerator cases.

data structures that encode special float values in Python such as `nan`. These values can be deserialized into Python values, but the procedure for this is beyond the scope of this paper.

Union types can be admitted to the embedding via the `register-union` API. Line 13 of Listing 8 contains an example of embedding a union of Python's integer, floating-point, and string types. 100 examples of this union type are then generated in the subsequent expression (line 16). The equivalent Python type annotations for this union are `int | float | str` and `Union[int, float, str]`.

7 Evaluation

With this embedding, our hope is to pave the way to formal reasoning about Python's types and to enable fuzzing and property-based testing of Python code. For our work to be suitable for these use cases, the examples of types that the embedding generates must be representative of values that actual Python code interacting with those types would expect. To verify that our embedding satisfies this criterion, we perform an evaluation of code coverage in four open source repositories: `mypy` [7], `mindsdb` [6], `black` [5], and `manticore` [4]. These repositories were chosen because they were noted as having a high

type annotation density in the type annotation study of Di Grazia and Pradel [16]. Our focal evaluation goal is to verify that we cover code that we expect to cover, given that we only have knowledge of the types of function signatures. Importantly, we expect *not* to cover code that has external file system dependencies, or code that has complex conditionals that block execution paths.

7.1 Setup

To prepare the repositories for fuzzing using the enumerative data types in our embedding, we extract type information from each codebase, and iteratively extend the embedding to embed as much type information in the codebase as possible with respect to the current limitations of our implementation. Recall that this procedure is given in Algorithm 1 and explained in Section 4. Further details of this information extraction are given in Xifaras's previous work [52].

Algorithm 2 Function Signature Extraction

Input: Type set S; set of function signatures in codebase F
Output: Set of fuzzable functions G

```
1: G ← ∅
2: for each f in F do
3:     if types(f) ⊆ S then
4:         G ← G ∪ {f}
5:     end if
6: end for
7: return G
```

After type registration is completed, *function signature extraction* can take place. The definition for this procedure is given in Algorithm 2. It iterates over all functions in the target codebase, and adds functions for which all types in the signature are present in the set of recognized types S. The helper function *types* is again used to extract this set from each signature.

The output of the function signature extraction step is a set of *appropriate functions*. In summary, a function is appropriate if its signature is fully annotated and every type that appears in the signature is embedded in ACL2s. These functions are then fuzzed in the manner described in the following subsection.

7.2 Experiment Design

After performing the previously described setup, we perform a small fuzz testing experiment on the set of appropriate functions with the intent of gathering code coverage information. We use the *coverage.py* [8] library to measure coverage. By default, this library measures line coverage (although it does not count whitespace lines, and it counts statements that wrap onto multiple lines as a single line).

We perform five independent trials, following evaluation guidance from Klees et al. [23]. In each trial, we fuzz each appropriate function using a stream of examples generated by the embedding's enumerators for 440 seconds. During fuzzing, the input-output samples are collected and stored, and "replayed" after fuzzing is complete to obtain code coverage.

7.3 Results

To provide insight into the coverage profile as the fuzz testing was taking place, we present coverage over time for the four repositories, averaged across the five independent trials. Figure 1 contains the

coverage results. Code coverage for a fuzzing trial in a repository is measured as the percentage of covered statements in the union of the sets of statements in the bodies of that repository's appropriate functions. The solid lines presented in Figure 1 represent the average of this coverage. The dashed and dotted lines above and below each solid indicate 95% confidence intervals for coverage. Note that the confidence interval lines are still being rendered for `mindsdb` and `manticore`, but they visually overlap with the solid average line.

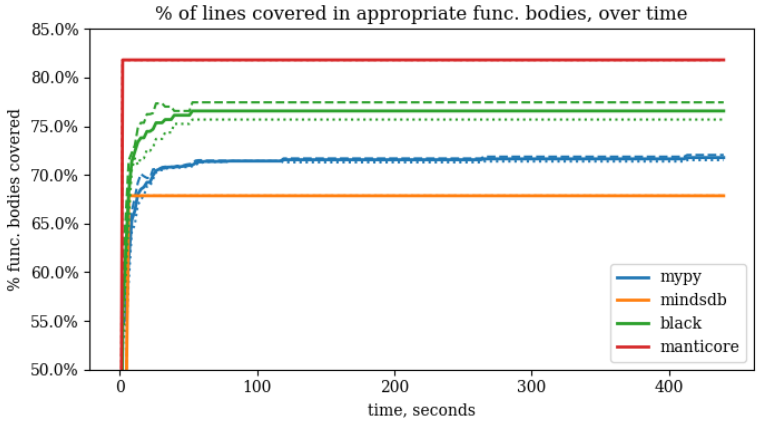

Figure 1: Results of coverage evaluation.

Coverage is generally obtained very quickly. Note also that `mypy` and `black` seem to have longer "knees" on their curves than `mindsdb` and `manticore`. An explanation for this is that, as shown in Table 4, the former repositories have significantly higher numbers of appropriate functions than the latter. This might introduce more variability into the results for the former repositories. Code coverage is also rather good. It ranges from about 68% to greater than 80%. Importantly, this code coverage is obtained without knowledge of the bodies of these functions, only their type signatures.

Repository	**Total Functions**	**Annotated Functions**	**Appropriate Functions**
mypy	1028	1028	132
mindsdb	400	55	5
black	248	248	35
manticore	211	26	5

Table 4: Function breakdown by repository.

Table 4 presents the breakdown of function totals across the four repositories studied. Importantly, a "function" in this context is a top-level function defined in Python. We do not currently consider functions that are defined as methods of classes. These are not counted in the totals, and they are not eligible to be appropriate functions. Python also supports the definition of nested functions. Functions that are nested within other functions are not counted in the total number of functions and are not eligible to be appropriate functions.

Although we obtained good coverage overall, there were certain function bodies for which we achieved lower coverage. Broadly, the reasons for lower coverage that we have observed can be bucketed into 1) overly broad type annotations that do not represent the data the function is expecting, 2) external

dependencies that the function has on either the file system or global program state, and 3) complex branch conditions that are hard to satisfy.

Figure 2 contains the *coverage.py* report output for one of the appropriate functions in `mypy` that was fuzzed, `infer_method_ret_type`. The low coverage here is because of the complex branch condition, which is checking whether the given string starts and ends with two underscore characters.

```
def infer_method_ret_type(name: str) -> str | None:
    """Infer return types for known special methods"""
    if name.startswith("__") and name.endswith("__"):
        name = name[2:-2]
        if name in ("float", "bool", "bytes", "int", "complex", "str"):
            return name
        # Note: __eq__ and co may return arbitrary types, but bool is good enough for stubgen.
        elif name in ("eq", "ne", "lt", "le", "gt", "ge", "contains"):
            return "bool"
        elif name in ("len", "length_hint", "index", "hash", "sizeof", "trunc", "floor", "ceil"):
            return "int"
        elif name in ("format", "repr"):
            return "str"
        elif name in ("init", "setitem", "del", "delitem"):
            return "None"
    return None
```

Figure 2: Complex string condition that is difficult to pass when the input can be any string.

Figure 3 shows an example of low coverage from the `black` repository which is not getting fully covered due to an unmet file system dependency. In this case, the argument to the function, `path_config`, represents a path to a valid TOML file. It is highly unlikely to spontaneously generate a valid file path, and we do not intentionally set up TOML files in a test bed for fuzzing. Therefore, execution results in an exception and the remainder of the function is not covered.

```
@mypyc_attr(patchable=True)
def parse_pyproject_toml(path_config: str) -> Dict[str, Any]:
    """Parse a pyproject toml file, pulling out relevant parts for Black.

    If parsing fails, will raise a tomllib.TOMLDecodeError.
    """
    pyproject_toml = _load_toml(path_config)
    config: Dict[str, Any] = pyproject_toml.get("tool", {}).get("black", {})
    config = {k.replace("--", "").replace("-", "_"): v for k, v in config.items()}

    if "target_version" not in config:
        inferred_target_version = infer_target_version(pyproject_toml)
        if inferred_target_version is not None:
            config["target_version"] = [v.name.lower() for v in inferred_target_version]

    return config
```

Figure 3: Unmet file system dependency causing function call to fail.

Figure 4a contains an example of low code coverage from `mindsdb`. This example represents a deficiency in the parameter type annotation of the function. The function expects that `'tree'` and `'pointer'` are both present as keys in the given dictionary, but the type annotation broadly specifies a dictionary with keys and values of any type. A `KeyError` exception is thrown that interrupts execution.

Finally, Figure 4b shows an example where we obtain full coverage of the function body. In this case, the enumerator for the `keys` argument, which is a list of strings, produces values that cover the three main branches in the function: keys having length 0, 1, or more than 1.

8 Discussion and Future Work

In this paper, we introduce an enumerative embedding of the Python type system in ACL2s. In our estimation, the principal application of the embedding, because it is enumerative, is fuzzing. Fuzzing

```
def _get_current_node(profiling: dict) -> dict:
    """ return the node that the pointer points to

        Args:
            profiling (dict): whole profiling data

        Returns:
            dict: current node
    """
    current_node = profiling['tree']
    for child_index in profiling['pointer']:
        current_node = current_node['children'][child_index]
    return current_node
```

(a) Dictionary lookup failed due to implicit dictionary structure.

```
def format_key_list(keys: list[str], *, short: bool = False) -> str:
    formatted_keys = [f'"{key}"' for key in keys]
    td = "" if short else "TypedDict "
    if len(keys) == 0:
        return f"no {td}keys"
    elif len(keys) == 1:
        return f"{td}key {formatted_keys[0]}"
    else:
        return f"{td}keys ({', '.join(formatted_keys)})"
```

(b) Function with no external dependencies and simple conditionals is fully covered.

Figure 4: Additional code coverage examples.

```
def create_decimal(whole_part: int, decimal_part: float) -> float:
    """Create a decimal number from the given whole part and decimal part"""
    return float(str(whole_part) + str(decimal_part).lstrip('0'))

def test_create_decimal_no_exception(x: int, y: float) -> bool:
    """Property-based test to ensure create_decimal doesn't throw (obviously, this fails)"""
    try:
        create_decimal(x, y)
    except:
        return False
    return True
```

Listing 9: Example of a property-based test that could be serviced by the enumerative embedding.

requires high-quality inputs to be successful, and the customizability of enumerators enables users to create representative examples of their data [52].

In light of its use as the foundation for fuzzing, we validated in our evaluation (Section 7) that the inputs generated by our custom enumerators cover code effectively in functions whose type signatures are embedded. However, we also found that code coverage is limited by type annotations that are too broad, external dependencies on program or system state, and complex branch conditions. These can be addressed in future work in the following ways:

1. **Overly broad annotations:** Analyze the errors that are raised when sending inputs into functions, or send in mock objects that are instrumented to track how they are used, to constrain the type definitions.
2. **External dependencies:** Implement mocks of global program state and the file system that the code being tested can interact with.
3. **Complex branch conditions:** Extract branch conditions, embed them in ACL2s, and use ACL2s to produce examples that satisfy and do not satisfy them.

A threat to the validity of these results is that the latest source code for the custom enumerators is different from the enumerators that were used to collect this data, but the adjustments are minor enough that we do not anticipate significant effects on the results.

The validity of the results and the embedding itself are also limited by the set of supported types. A core set of types has been implemented, but there are many types in Python's `typing` module, such as

`Sequence` and `Iterable`, that are used often in Python code. In particular, function types are important because functions are first-class objects in Python. They are denoted in type annotations using the `Callable` annotation in the `typing` module. This embedding becomes significantly more usable on the average repository when these types are embedded. This is a top priority for future work.

Given its suitability for fuzzing, this embedding further enables property-based testing for typed properties written as Python functions. For instance, Listing 9 specifies the property "`create_decimal()` does not throw an exception." Examples for Python integers and floats generated by enumerators can be streamed as inputs to `test_create_decimal_no_exception`, and if this function returns False, the property is violated. This testing methodology represents a practical compromise between unit testing, where fixed scenarios with strong assumptions are tested, and formal verification. Property-based testing libraries exist for Python [30, 19], and we look forward to evaluating opportunities for integration and collaboration in future work.

Another compelling application for this embedding is type checking. Once the full type system semantics are embedded and additional information about the code such as basic control flow skeletons is extracted, a theorem-prover-based type checker could be built which may have better soundness and completeness properties than current solutions. We leave this as another exciting direction for future work.

9 Conclusion

As Python continues to grow in popularity, the ability to test applications written in it grows in importance. Meanwhile, the growing prevalence of developer-added type annotations in Python renders automated analyses more tractable [16]. In this paper, we enable tool developers to leverage this situation with an embedding of the Python type system in ACL2s. This embedding is enumerative, meaning that examples of types can be generated easily. This enables dynamic testing of Python code, which is useful in the absence of a formal model of Python's complex semantics. The embedding is additionally extensible. We invite the community to extend this embedding with additional types and typing constructs. Python's documentation contains information on what types and typing constructs are available in the language [37].

Our long-term goal is to create data definitions and enumerators that support the entire type system. Eventually, this embedding may serve as a foundation for advanced property-based testing and reasoning in Python, and we hope to advance further toward this vision in future work.

Acknowledgments

We would like to acknowledge the MIT SuperCloud team for granting us access to their high-performance computing environment on which we ran our experiments [40].

References

[1] *pickle — Python object serialization*. Available at `https://docs.python.org/3/library/pickle.html`.

[2] (2015): *pytype*. Available at `https://github.com/google/pytype`.

[3] (2022): *The top programming languages*. Available at `https://octoverse.github.com/2022/top-programming-languages`.

[4] (2023): *Manticore*. Available at https://github.com/trailofbits/manticore.

[5] (2025): *Black: The Uncompromising Code Formatter*. Available at https://github.com/psf/black.

[6] (2025): *mindsdb*. Available at https://github.com/mindsdb/mindsdb.

[7] (2025): *Mypy: Static Typing for Python*. Available at https://github.com/python/mypy.

[8] Ned Batchelder (2023): *Coverage.py*. Available at https://coverage.readthedocs.io/en/7.1.0/.

[9] 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.

[10] Harsh Raju Chamarthi (2016): *Interactive Non-theorem Disproving*. Ph.D. thesis, Northeastern University, doi:10.17760/D20467205.

[11] 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.

[12] 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*, doi:10.4204/EPTCS.152.3.

[13] 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.

[14] Anna van Deusen (2023): *Python Popularity: The Rise of a Global Programming Language*. Available at https://flatironschool.com/blog/python-popularity-the-rise-of-a-global-programming-language/.

[15] 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.

[16] Luca Di Grazia & Michael Pradel (2022): *The evolution of type annotations in python: an empirical study*. In Abhik Roychoudhury, Cristian Cadar & Miryung Kim, editors: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, ACM, pp. 209–220, doi:10.1145/3540250.3549114.

[17] Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter & Andreas Stefik (2014): *An empirical study on the impact of static typing on software maintainability*. *Empirical Software Engineering* 19(5), pp. 1335–1382, doi:10.1007/s10664-013-9289-1.

[18] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke & Travis E. Oliphant (2020): *Array programming with NumPy*. *Nature* 585(7825), pp. 357–362, doi:10.1038/s41586-020-2649-2.

[19] Zac Hatfield-Dodds (2022): *HypoFuzz*. Available at https://hypofuzz.com/.

[20] Matt Kaufmann, Panagiotis Manolios & J Strother Moore (2000): *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, doi:10.1007/978-1-4615-4449-4.

[21] Matt Kaufmann, Panagiotis Manolios & J Strother Moore (2000): *Computer-Aided Reasoning: Case Studies*. Kluwer Academic Publishers, doi:10.1007/978-1-4757-3188-0.

[22] Matt Kaufmann & J Strother Moore (2025): *ACL2 homepage*. Available at https://www.cs.utexas.edu/users/moore/acl2/.

[23] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei & Michael Hicks (2018): *Evaluating Fuzz Testing*. In David Lie, Mohammad Mannan, Michael Backes & XiaoFeng Wang, editors: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, ACM, pp. 2123–2138, doi:10.1145/3243734.3243804.

[24] Maximilian A. Köhl (2021): *An Executable Structural Operational Formal Semantics for Python*. arXiv:2109.03139.

[25] Ankit Kumar & Panagiotis Manolios (2021): *Mathematical Programming Modulo Strings*. In: *Formal Methods in Computer Aided Design, FMCAD 2021, New Haven, CT, USA, October 19-22, 2021*, IEEE, pp. 261–270, doi:10.34727/2021/ISBN.978-3-85448-046-4_36.

[26] Jukka Lehtosalo (2019): *Our journey to type checking 4 million lines of Python*. Available at `https://dropbox.tech/application/our-journey-to-type-checking-4-million-lines-of-python`.

[27] Jukka Lehtosalo (2024): *Mypy Documentation*. Available at `https://mypy.readthedocs.io/_/downloads/en/stable/pdf/`.

[28] Ivan Levkivskyi, Jukka Lehtosalo & Łukasz Langa (2017): *PEP 544 - Protocols: Structural Subtyping (static duck typing)*. Available at `https://peps.python.org/pep-0544/`.

[29] Wen Li, Haoran Yang, Xiapu Luo, Long Cheng & Haipeng Cai (2023): *PyRTFuzz: Detecting Bugs in Python Runtimes via Two-Level Collaborative Fuzzing*. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers & Engin Kirda, editors: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, ACM, pp. 1645–1659, doi:10.1145/3576915.3623166.

[30] David MacIver & Zac Hatfield-Dodds (2019): *Hypothesis: A new approach to property-based testing*. *J. Open Source Softw.* 4(43), p. 1891, doi:10.21105/JOSS.01891.

[31] 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.

[32] 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.

[33] 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.

[34] Panagiotis Manolios & Daron Vroon (2006): *Termination Analysis with Calling Context Graphs*. In Thomas Ball & Robert B. Jones, editors: *Computer Aided Verification*, *Lecture Notes in Computer Science* 4144, Springer, pp. 401–414, doi:10.1007/11817963_36.

[35] Joe Gibbs Politz, Alejandro Martinez, Mae Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu & Shriram Krishnamurthi (2013): *Python: the full monty*. In Antony L. Hosking, Patrick Th. Eugster & Cristina V. Lopes, editors: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, ACM, pp. 217–232, doi:10.1145/2509136.2509536.

[36] Python Software Foundation (2025): *Built-in Types*. Available at `https://docs.python.org/3/library/stdtypes.html`.

[37] Python Software Foundation (2025): *typing – Support for type hints*. Available at `https://docs.python.org/3/library/typing.html`.

[38] Ingkarat Rak-amnouykit, Daniel McCrevan, Ana L. Milanova, Martin Hirzel & Julian Dolby (2020): *Python 3 types in the wild: a tale of two type systems*. In Matthew Flat, editor: *DLS 2020: Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages, Virtual Event, USA, November 17, 2020*, ACM, pp. 57–70, doi:10.1145/3426422.3426981.

[39] James Franklin Ranson (2008): *A semantics of Python in Isabelle/HOL*. Faculty of Graduate Studies and Research, University of Regina.

[40] Albert Reuther, Jeremy Kepner, Chansup Byun, Siddharth Samsi, William Arcand, David Bestor, Bill Bergeron, Vijay Gadepally, Michael Houle, Matthew Hubbell, Michael Jones, Anna Klein, Lauren Milechin, Julia Mullen, Andrew Prout, Antonio Rosa, Charles Yee & Peter Michaleas (2018): *Interactive supercomputing on 40,000 cores for machine learning and data analysis*. In: *2018 IEEE High Performance extreme Computing Conference (HPEC)*, IEEE, pp. 1–6, doi:10.1109/HPEC.2018.8547629.

[41] Guido van Rossum, Jukka Lehtosalo & Łukasz Langa (2014): *PEP 484 - Type Hints*. Available at `https://peps.python.org/pep-0484/`.

[42] Guido van Rossum & Ivan Levkivskyi (2014): *PEP 483 - The Theory of Type Hints*. Available at `https://peps.python.org/pep-0483/`.

[43] Ori Roth (2022): *Python Type Hints are Turing Complete*. *CoRR* abs/2208.14755, doi:10.48550/ARXIV.2208.14755. arXiv:2208.14755.

[44] Gideon Joachim Smeding (2009): *An executable operational semantics for Python*. Universiteit Utrecht.

[45] Evan Sultanik (2021): *Never a dill moment: Exploiting machine learning pickle files*. Available at `https://blog.trailofbits.com/2021/03/15/never-a-dill-moment-exploiting-machine-learning-pickle-files/`.

[46] the mypy project (2014): *mypy*. Available at `https://mypy-lang.org/`.

[47] Andrew T. Walter, Benjamin Boskin, Seth Cooper & Panagiotis Manolios (2019): *Gamification of Loop-Invariant Discovery from Code*. In Edith Law & Jennifer Wortman Vaughan, editors: *Proceedings of the Seventh AAAI Conference on Human Computation and Crowdsourcing, HCOMP 2019, Stevenson, WA, USA, October 28-30, 2019*, AAAI Press, pp. 188–196, doi:10.1609/HCOMP.V7I1.5277.

[48] Andrew T. Walter, David A. Greve & Panagiotis Manolios (2022): *Enumerative Data Types with Constraints*. In Alberto Griggio & Neha Rungta, editors: *22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022*, IEEE, pp. 189–198, doi:10.34727/2022/ISBN.978-3-85448-053-2_25.

[49] Andrew T. Walter, Ankit Kumar & Panagiotis Manolios (2023): *Proving Calculational Proofs Correct*. In Alessandro Coglio & Sol Swords, editors: *Proceedings of the 18th International Workshop on the ACL2 Theorem Prover and Its Applications*, *EPTCS* 393, pp. 133–150, doi:10.4204/EPTCS.393.11.

[50] 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.

[51] Andrew T. Walter & Panagiotis Manolios (2025): *An ACL2s Interface to Z3*. *Electronic Proceedings in Theoretical Computer Science* 423, Open Publishing Association, pp. 104–123, doi:10.4204/EPTCS.423.10.

[52] Samuel Xifaras (2024): *Leveraging Type Annotations for Effective Fuzzing of Python Programs*. Master's thesis, Northeastern University, doi:10.17760/D20705205.

[53] Hongyu Zhai, Casey Casalnuovo & Prem Devanbu (2019): *Test Coverage in Python Programs*. In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, IEEE, Montreal, QC, Canada, pp. 116–120, doi:10.1109/MSR.2019.00027.