EPTCS 192

Proceedings of the

Thirteenth International Workshop on the ACL2 Theorem Prover and Its Applications

Austin, Texas, USA, 1-2 October 2015

Edited by: Matt Kaufmann and David L. Rager

Published: 18th September 2015
DOI: 10.4204/EPTCS.192
ISSN: 2075-2180
Open Publishing Association

Table of Contents

Preface

This volume contains the proceedings of the Thirteenth International Workshop on the ACL2 Theorem Prover and Its Applications, ACL2 2015, a two-day workshop held in Austin, Texas, USA, on October 1-2, 2015. ACL2 workshops occur at approximately 18-month intervals and provide a major technical forum for researchers to present and discuss improvements and extensions to the theorem prover, comparisons of ACL2 with other systems, and applications of ACL2 in formal verification.

ACL2 is a state-of-the-art automated reasoning system that has been successfully applied in academia, government, and industry for specification and verification of computing systems and in teaching computer science courses. In 2005, Boyer, Kaufmann, and Moore were awarded the 2005 ACM Software System Award for their work on ACL2 and the other theorem provers in the Boyer-Moore family.

The proceedings of ACL2 2015 include the eight technical papers and one extended abstract that were presented at the workshop. Each submission received at least three reviews. The workshop also included two invited talks: *Lessons Learned over 45 Years in Theorem Proving*, by J Strother Moore, and *Verification in the Age of Integration*, by John O'Leary. The workshop also included several *rump sessions* discussing ongoing research and a panel discussion about the use of ACL2 within industry.

We thank the members of the Program Committee and their sub-reviewers for providing careful and detailed reviews of all the papers. We thank the members of the Steering Committee for their help and guidance. We thank EasyChair for the use of its excellent conference management system. We thank EPTCS and the arXiv for publishing the workshop proceedings in an open-access format.

Matt Kaufmann and David L. Rager
October 2015

Program Chairs

Matt Kaufmann
David L. Rager

Program Committee

Rob Arthan	Lemma 1 Ltd.
Jared Davis	Centaur Technology Inc.
Ruben Gamboa	University of Wyoming
David Greve	Rockwell Collins Inc.
David Hardin	Rockwell Collins Inc.
Marijn Heule	The University of Texas at Austin
Warren A. Hunt, Jr.	The University of Texas at Austin
Matt Kaufmann	The University of Texas at Austin
Panagiotis Manolios	Northeastern University
Francisco Jesús Martín Mateos	University of Sevilla
John O'Leary	Intel Corporation
Sam Owre	SRI International
David L. Rager	Oracle Corporation
Sandip Ray	Intel Corporation
Julien Schmaltz	Eindhoven University of Technology
Anna Slobodova	Centaur Technology Inc.
Eric Smith	Kestrel Institute
Freek Verbeek	Open University of The Netherlands

Additional Reviewers

Coglio, Alessandro

A Brief Introduction to Oracle's Use of ACL2 in Verifying Floating-point and Integer Arithmetic

David L. Rager, Jo Ebergen, Austin Lee, Dmitry Nadezhin, Ben Selfridge
Oracle
{david.rager,jo.ebergen,austin.lee,dmitry.nadezhin,ben.selfridge}@oracle.com

Cuong K. Chau
The University of Texas at Austin
ckcuong@cs.utexas.edu

Oracle has developed new implementations for floating-point division and square root and integer division. The Oracle implementations are a variant of the Goldschmidt algorithm [3], an algorithm that serves as the basis for a different AMD K7 implementation [2] and other processors. Our task was to verify the correctness of these implementations all the way down to the Verilog by showing bit-for-bit equivalence with the IEEE 754 Standard on floating-point arithmetic and the integer divide specification in the SPARC ISA. Performing such verifications involved many steps:

- Parsing the Verilog and bringing the design into the ACL2 logic – *parsing*,
- Abstracting low-level bit-oriented primitives, (e.g., *nand*s, *nor*s, muxes, etc.) to higher-level data types and mathematical operations like addition and multiplication – *algorithm extraction*, and
- Proving that sequences of these mathematical operations satisfy the IEEE 754 and Integer Divide specifications – *algorithm verification*.

This work is similar in spirit to work done by AMD [2], Centaur [4], and others.

1 Parsing

We wanted to make as few assumptions as reasonably possible, and we wanted as much of what we did to be mechanically checked. As such, it was important to reason directly about the Verilog – not just an abstraction of the Verilog created by hand that would then have to be maintained every time the Verilog changed. If the implementation changes in a subtle and incorrect way, we wanted our proofs to break!

We used the ACL2 System and Libraries [1] to symbolically simulate the circuit and verify its correctness. Specifically, we used the GL system to quickly reason about finite objects (i.e., bit vectors). We used the VL 2014 Parser to parse the Verilog, and we used Esim to determine the semantics of the resulting parse trees. Finally, we used the Symbolic Test Vector (STV) framework to initialize values of the circuit and provide the timing abstractions necessary to reason about values that the circuit receives as input and returns as output.

2 Algorithm Extraction

The Goldschmidt algorithm consists of a repetition of multiplications, additions, and bit-wise complements. We wanted the person verifying the Goldschmidt algorithm against our specifications to be able to reason in terms of these high-level primitives (as opposed to reasoning about `nand`'s and `nor`'s). As

M. Kaufmann and D. Rager (Eds.): ACL2 Workshop 2015 (ACL2 2015).
EPTCS 192, 2015, pp. 1–2, doi:10.4204/EPTCS.192.1

such, we proved that various compositions of low-level operations implement these higher-level mathematical operations. We found the GL system to be very helpful in many of these proofs, but GL (really, BDDs and SAT solvers) could not automatically prove all of the necessary abstractions. For example, we used traditional ACL2 rewriting to reason about the composition of Booth encoders and CSA trees to implement multiplication.

3 Algorithm Verification

Our final task was to show that the aforementioned series of mathematical operations implemented the IEEE 754 specification of floating-point division and square root and our specification for integer division. This first required writing such specifications, which we validated against millions of concrete test vectors. We then proved that the series of mathematical operations from the algorithm extraction implement these specifications. This proof was a sizable effort.

4 Results

At the end of it all, we have proved, under some assumptions, that the Verilog matches bit for bit with our extracted algorithm. We have also proved that the extracted algorithm satisfies the IEEE floating-point division and square root specifications and the integer division specifications. Thus, transitively, we know that the Verilog implements these specifications.

We discovered no errors with respect to the complete IEEE specification (including exceptions, denormals, and special values) in the Verilog design. As a result, we know, for example, that we have neither omitted an entry from our lookup table, nor accidentally fused together a wire in our multiplier.

Despite the absence of RTL errors, this verification effort was still necessary because there was simply no other way to obtain a reasonable amount of coverage. Another benefit of our work was the discovery of several optimizations, which were all implemented and proven correct. Our thorough error analysis led to an optimization in the lookup tables for division and square root, yielding a reduction of 75% and 50% in the two lookup tables, respectively. Other optimizations led to simplifications in the hardware and in the proof. We look forward to reporting on these at a later date.

5 Acknowledgements

We thank Jeff Brooks, Greg Grohoski, Warren Hunt, Matt Kaufmann, Govind Murugan, Chris Olson, and Greg A. Smith for their technical help and support.

References

[1] ACL2 (2015): *ACL2 Documentation*. Http://www.cs.utexas.edu/users/moore/acl2/v7-1/combined-manual/.

[2] S.F. Oberman (1999): *Floating point division and square root algorithms and implementation in the AMD-K7TM microprocessor*. In: *Computer Arithmetic, 1999. Proceedings. 14th IEEE Symposium on*, pp. 106–115, doi:10.1109/ARITH.1999.762835.

[3] S.F. Oberman & M. Flynn (1997): *Division algorithms and implementations*. *Computers, IEEE Transactions on* 46(8), pp. 833–854, doi:10.1109/12.609274.

[4] A. Slobodova, J. Davis, S. Swords & W. Hunt (2011): *A flexible formal verification framework for industrial scale validation*. In: *Formal Methods and Models for Codesign (MEMOCODE), 2011 9th IEEE/ACM International Conference on*, pp. 89–97, doi:10.1109/MEMCOD.2011.5970515.

Fix Your Types

Sol Swords Jared Davis
Centaur Technology, Inc.
7600-C N. Capital of Texas Hwy, Suite 300
Austin, TX 78731
{sswords,jared}@centtech.com

When using existing ACL2 datatype frameworks, many theorems require type hypotheses. These hypotheses slow down the theorem prover, are tedious to write, and are easy to forget. We describe a principled approach to types that provides strong type safety and execution efficiency while avoiding type hypotheses, and we present a library that automates this approach. Using this approach, types help you catch programming errors and then get out of the way of theorem proving.

1 Introduction

ACL2 is often described as "untyped," and this is certainly true to some degree. Terms like `(+ 0 "hello")`, which would be not be accepted by the static type checks of languages like Java, are legal and well-defined in the logic, and can even be executed so long as guard checking is disabled. Terms like `(/ 5 0)`, which would be well-typed but would cause a run-time error in most programming languages, are also logically well-defined and can also be executed when guards are not checked.

Of course, most ACL2 code is written with particular types in mind, often expressed as the guards of the functions. When proving properties of such code, it's easy to get tripped up by corner cases where some variables of the theorem are of the wrong types. To avoid this, one strategy is to begin a theorem with a list of type hypotheses, one for each variable mentioned. These hypotheses act as a kind of insurance: we may not know whether they're necessary or not, but including them might save us from having to debug failed proofs caused by missing type assumptions.

On the other hand, lists of type hypotheses are often repetitive, take time to write, and make the formulas we're proving larger and less elegant. We can hide much of this with macros, e.g., the `define` utility has special options like `:hyp :guard` for including the guard as a hypothesis in return-value theorems. But even then, these hypotheses will cause extra work for the rewriter since it must relieve them before it can apply our theorem, and may make it harder to carry out later proofs since our theorem will not be applied unless its hypotheses can be relieved.

Accordingly, after we have proven a theorem, a good practice is to try to "strengthen" it by removing any unnecessary hypotheses. There is even a tool, `remove-hyps`, that tries to automatically identify unnecessary hypotheses ex post facto. While strengthening theorems is useful, it is limited. For instance, we (of course) cannot eliminate type hypotheses that are actually necessary for the formula to be a theorem. It can also be tedious, e.g., automation such as `define`'s `:hyp :guard` does not provide a convenient way to remove parts of the guard. This is not a purely theoretical concern; see for instance ACL2 Issue 167, a request for such a feature.

An alternate strategy, which is well-known and certainly not novel, is to more carefully code our functions so that they always treat ill-typed inputs according to some particular *fixing convention*. By following this approach, we can typically avoid the need for type hypotheses altogether. Many examples of this approach can be found throughout ACL2. To name a few:

M. Kaufmann and D. Rager (Eds.): ACL2 Workshop 2015 (ACL2 2015).
EPTCS 192, 2015, pp. 3–16, doi:10.4204/EPTCS.192.2

- Arithmetic functions treat non-numbers as 0; functions expecting a particular type of number (integer, natural) treat anything else as 0—e.g., `zp`, `nth`, `logbitp`.
- Functions expecting a string treat a non-string as `""`—e.g., `char`, `(coerce x 'list)`.
- Any atom is treated as `nil` by `car`, `cdr`, `endp`, etc.
- The `std/osets` [5] library functions treat non-sets as `nil`.

Following this strategy typically takes a small bit of initial setup, e.g., to agree upon and implement a fixing convention. But once this convention is in place, type hypotheses can be eliminated from many theorems. For instance, we have unconditional theorems such as $a + b = b + a$ without hypotheses about a and b being numbers, and $X \subseteq X$ without hypotheses about X being a set. By eliminating type hypotheses, these theorems become easier to read and write, and can be more efficiently and reliably used to simplifying later proof goals.

Unfortunately, existing datatype definition frameworks for ACL2 don't provide any easy way to follow the fixing strategy. For example, consider the available macros for introducing product types, like `defstructure` [3], `defdata` [4], and `defaggregate`. These macros define constructors and accessor functions that do not support any particular convention for dealing with ill-typed fields or products. Consider a simple `student` structure introduced by:

```
(defaggregate student
  ((name stringp)
   (age  natp)))
```

The constructor and accessors for `student` structures will have strong guards that are useful for revealing programming errors in function definitions. However, in the logic, nothing prevents us from invoking the `student` constructor on ill-typed arguments. For instance, in cases like:

```
(make-student :name 6 :age "Calista")
```

the constructor fails to produce a valid `student-p`. The accessors suffer from similar problems, for instance the following term is equal to `6`, which is not a well-typed student name:

```
(student->name (make-student :name 6 :age "Calista"))
```

Consequently, reasoning about these structures almost always requires type hypotheses. Since the types defined by these frameworks are often found at the lowest levels of ACL2 models, these hypotheses percolate upwards, infecting the entire the code base!

In this paper, we address this problem with the following contributions:

- We present, in precise terms, a **fixtype discipline** for working with types in ACL2 (Section 2). This discipline allows efficient reasoning via avoiding type hypotheses, strong type checking via ACL2's guard mechanism, and preserves efficient execution via `mbe`.
- Manually following the fixtype discipline would be tedious. Accordingly, we present a new library, `FTY` (short for "fixtypes"), which provides automation for following the discipline (Section 3). The FTY library contains tools that automate the introduction of new types and assist with creating functions that "properly" operate on those types.

While there is room for improvement (Section 4), the approach and automation that we present is practical and scales up to complex modeling efforts. We have successfully used FTY as the type system for two large libraries: VL, which processes Verilog and SystemVerilog source code; and SV, a hardware modeling and analysis framework. VL, in particular, involves a very complex hierarchy of types. For instance, it includes a 30-way mutually recursive datatype that represents SystemVerilog expressions, types, and related syntactic constructs.

2 The Fixtype Discipline

We begin, in this section, by describing in precise terms a *fixtype discipline* for working with types in ACL2. In our experience, following this discipline is an effective way to obtain the benefits of strong type checking while keeping types out of the way of theorem proving.

The basic philosophy behind the fixtype discipline is that all functions that take inputs of a particular type should treat any inputs that are *not* of that type in a consistent way. This can be done using *fixing functions*.

Definition 1. A **fixing function** *fix* for a (unary) type predicate *typep* is a (unary) function that (1) always produces an object of that type, and (2) is the identity on any object of that type. That is, it satisfies:

$$(1) \quad \forall x : typep(fix(x))$$
$$(2) \quad \forall x : typep(x) \Rightarrow fix(x) = x$$

Given a fixing function, an easy way to ensure that some new definition treats all of its inputs in a type-consistent way is to immediately apply the appropriate fixing function to each input before proceeding with the main body of the function. For guard-verified functions, this preliminary fixing can be done for free using `mbe`. Alternatively, if all occurrences of an input variable in the function's body occur in contexts that are already type-consistent, then explicit fixing isn't necessary.

When a function follows this approach, the fixing functions become "transparent" to that function. For instance, since `nth` properly fixes its index argument to a natural number, the following holds:

```
(defthm nth-of-nfix
  (equal (nth (nfix n) x)
         (nth n x)))
```

Given any fixing function, we can define a corresponding equivalence relation. For instance, for naturals, we can define `nat-equiv` as equality up to `nfix`:

```
(defun nat-equiv (x y)
  (equal (nfix x) (nfix y)))
```

Functions that properly fix their arguments will satisfy a congruence for this equivalence under equality: that is, they produce `equal` results when given equivalent arguments. For instance, for `nth`:

```
(defthm nat-equiv-congruence-for-nth
   (implies (nat-equiv n m)
            (equal (nth n x)
                   (nth m x)))
  :rule-classes :congruence)
```

We can now define our fixtype discipline.

Definition 2. A function follows the **fixtype discipline** if, for each typed input, the type has a corresponding fixing function and equivalence relation, and the function produces equal results given type-equivalent inputs.

A consequence of following the fixtype discipline is that theorems can avoid type hypotheses.

Theorem 1. *Let typep be a type and let $\equiv$ be the equivalence relation induced by a fixing function for typep. Let $C(x)$ be a conjecture satisfying the congruence*

$$(x \equiv x') \Rightarrow \left(C(x) \Leftrightarrow C\left(x'\right)\right).$$

Then $C(x)$ is a theorem if and only if $typep(x) \Rightarrow C(x)$ is a theorem.

This congruence means that $C(x)$ is a formula where the variable x is consistently treated according to the fixing discipline for *typep*; in this case, it isn't necessary to include $typep(x)$ as a hypothesis. This generalizes easily to additional variables.

3 The FTY Library

If we want to follow the fixtype discipline, all of our type predicates need to have corresponding fixing functions and equivalence relations. Also, as we introduce new functions that operate on these types, we need to prove that these functions satisfy the appropriate congruences, i.e., that they treat their inputs in a type-consistent way.

Although these definitions and proofs are straightforward, it would be very tedious to carry them out manually. It would also be difficult to make use of libraries like `std/util` or `defdata` since the functions these frameworks introduce do not follow the fixtype discipline. This is unfortunate because these libraries really make it far more convenient to introduce new types.

To address this, we have developed a new library, named FTY, which provides several utilities to automate this boilerplate work and to facilitate the introduction of new types that follow the discipline. Among these utilities, we have:

- `deffixtype`, which associates a type predicate with a fixing function and equivalence relation, for defining base types like `natp`, `stringp`, and custom user-defined base types. (Section 3.1)
- `deftypes` and associated utilities `defprod`, `deftagsum`, `deflist`, `defalist`, and more, which define new derived fixtype-compliant product types, sum types, list types, etc. (Section 3.2)
- `deffixequiv` and `deffixequiv-mutual`, which prove the appropriate type congruences for functions that operate on these types. (Section 3.3)

We now briefly describe these utilities. We focus here on what these utilities automate and how this helps to make it easier to follow the fixtype discipline. More detailed information on how to practically make use of these utilities, their available options, etc., can be found in the FTY documentation [8] in the ACL2+Books Manual.

3.1 Deffixtype

The FTY library uses an ACL2 `table` to record the associations between the name, predicate, fixing function, and equivalence relation for each known type. This information is used by many later FTY utilities to improve automation. For instance, when we define a new structure, this table allows us to look up the right fixing function and equivalence relation to use for each field just by its type, without needing to be repetitively told the fixing function and equivalence relation for every field.

The **deffixtype** utility is used to register new *base types*, i.e., types that are not defined in terms of other FTY types, with this table. Here is an example, which registers a new type named `nat`, recognized by `natp`, with fixing function `nfix`, and with the equivalence relation `nat-equiv`:

```
(deffixtype nat
  :pred  natp
  :fix   nfix
  :equiv nat-equiv)
```

The type name does not need to be a function name; we typically use the name of the predicate without the final "p" or "-p." The predicate and fixing function must always be provided by the user and defined ahead of time. `Deffixtype` can automatically define the equivalence relation based on the fixing function, or it can use an existing equivalence relation.

We usually do not need to invoke `deffixtype` directly. FTY includes a `basetypes` book that sets up these associations for basic ACL2 types like naturals, integers, characters, Booleans, strings, etc. When new derived types are introduced by FTY macros like `deftypes` (Section 3.2), they are automatically registered with the table. On the other hand, `deffixtype` is occasionally useful for defining low-level custom base types, or types that use special encodings, or that for some other reason we prefer not to introduce with `deftypes`.

Choosing a good fixing function for a type is not always straightforward. As far as the fixtype discipline and the FTY library is concerned, any function that satisfies Definition 1 suffices. However, the way in which ill-typed objects are mapped into the type affects which functions will have proper congruences for the induced equivalence relation. Some choices are dictated by pre-existing ACL2 conventions; for example, if we wrote our own `my-nat-fix` function that coerced non-naturals to 5 instead of 0, then this fixing function wouldn't be transparent to built-in functions such as `zp` and `nth`.

The fixing function's guard may optionally require that the input object already be of the type. This allows the fixing function to be coded so that it is essentially free to execute, using `mbe` so that the executable body is just the identity. It is also generally useful to inline the fixing function to avoid the small overhead of a function call. For example:

```
(defun-inline string-fix (x)
  (declare (xargs :guard (stringp x)))
  (mbe :logic (if (stringp x) x "")
       :exec x))
```

3.2 Deftypes and Supporting Utilities

Whereas `deffixtype` is useful for registering base types and special, custom types, the **deftypes** suite of tools can be used to easily define common kinds of derived types. The constructors, accessors, and other supporting functions introduced for these types follow the fixtype discipline, and the new types are automatically registered with `deffixtype`. There are utilities for introducing many kinds of types:

- `defprod`, which defines a product type,
- `deftagsum`, which defines a tagged sum of products,
- `deflist`, which defines a list type which has elements of a given type,
- `defalist`, which defines an alist type with keys and values of given types,
- `defoption`, which defines an option/maybe type,
- and others.

Using these macros is not much different than using other data definition libraries. For instance, we can introduce a basic `student` structure as follows:

```
(defprod student
  ((name stringp)
   (age  natp)))
```

This is very much like introducting a structure with `defaggregate`: it produces a recognizer, constructor, accessors for the fields, `b*` binders, and readable make/change macros. Unlike `defaggregate`, it also generates a fixing function, `student-fix`, an equivalence relation, `student-equiv`, and registers the new student type with `deffixtype`. The constructor and accessor functions for the new type also follow the fixtype discipline, e.g., we unconditionally have theorems such as:

- `(student-p (student name age))`
- `(stringp (student->name x))`
- `(natp (student->age x))`

A notable feature of `deftypes` is that it also provides strong support for mutually recursive types. In particular, several calls of utilities such as `defprod`, `deflist`, etc., may be combined inside a `deftypes` form to create a mutually-recursive clique of types. For example, to model a simple arithmetic term language such as:

$$\begin{array}{rcl} \text{aterm} & = & \mathsf{Num}\ \{\texttt{val} :: \text{integer}\} \\ & | & \mathsf{Sum}\ \{\texttt{args} :: \text{List aterm}\} \\ & | & \mathsf{Minus}\ \{\texttt{arg} :: \text{aterm}\} \end{array}$$

We might write the following `deftypes` form:

```
(deftypes arithmetic-terms
  (deftagsum aterm
    (:num   ((val integerp)))
    (:sum   ((args atermlist)))
    (:minus ((arg aterm))))
  (deflist atermlist
    :elt-type aterm))
```

As you might expect, this form creates the basic predicates, fixing functions, and equivalence relations for `aterm`s that are needed for the fixtype discipline:

- Predicates `aterm-p` and `atermlist-p`,
- Fixing functions `aterm-fix` and `atermlist-fix`, and
- Equivalence relations `aterm-equiv` and `atermlist-equiv`.

It also registers the new types with `deffixtype`. The form also defines several functions and tools for working with these new types, all of which have appropriate congruences for the fixtype discipline:

- A kind function, `aterm-kind`, to determine the kind of an `aterm`, e.g., `:num`, `:sum`, or `:minus`.
- Constructors for each kind of `aterm`: `aterm-num`, `aterm-sum`, and `aterm-minus`, and associated make/change macros in the style of `defaggregate`/`defprod`.
- Accessors for each kind of `aterm`: `aterm-num->val`, `aterm-sum->args`, `aterm-minus->arg` and associated `b*` binders.

- Measure functions, `aterm-count` and `atermlist-count`, appropriate for structurally recurring over objects of these types.

For convenience, a macro `aterm-case` is also introduced. This macro allows us to implement the common coding scheme of cases on the kind of an `aterm`, followed by binding variables to any needed fields of the product. Here is a simple example of using `aterm` structures.

```
(defines aterm-eval
  (define aterm-eval ((x aterm-p))
    :measure (aterm-count x)
    :returns (val integerp)
    :verify-guards nil
    (aterm-case x
      :num x.val
      :sum (atermlist-sum x.args)
      :minus (- (aterm-eval x.arg))))

  (define atermlist-sum ((x atermlist-p))
    :measure (atermlist-count x)
    :returns (val integerp)
    (if (atom x)
        0
      (+ (aterm-eval (car x))
         (atermlist-sum (cdr x)))))
  ///
  (verify-guards aterm-eval))
```

3.3 Deffixequiv and Deffixequiv-mutual

Together, `deffixtype` and `deftypes` allow us to largely automate the process of introducing new types that support the fixtype discipline. But this is only half the battle. When we define new functions that make use of these types, we are still left with the task of proving that these functions satisfy the appropriate congruences for every argument of these types. If our model or program involves many function definitions, this can be a lot of tedious work.

To automate this process, FTY offers two related utilities, **deffixequiv** and **deffixequiv-mutual**. These utilities are integrated with `define` and `defines` and also make use of the table of types from `deffixtype`. This allows them to figure out what theorems are needed, often without any help at all. In particular, the types of the arguments are inferred from the extended formals of the each function. The corresponding fixing functions and equivalence relations can then be looked up from the table, and the appropriate congruence rules can be generated. Besides congruence rules, we additionally generate rules that normalize constant arguments to their type-fixed forms.

Consider the `aterm-eval` example above. To generate the congruence rules for both `aterm-eval` and `atermlist-eval`, it suffices to invoke:

```
(deffixequiv-mutual aterm-eval)
```

The `deffixequiv-mutual` macro determines the types of the arguments by examining the guards specified in the `define` formals, and it uses the flag induction scheme produced by `defines` to automatically prove the congruence.

For recursive or mutually-recursive functions, proving a congruence directly can be difficult because there are two calls of the function in the statement of the theorem, and these two calls may suggest different induction schemes that may not be simple to merge. However, the congruences we are concerned with follow from the fact that the fixing function is transparent to the function, which can usually be proved straightforwardly by induction on the function's own recursion scheme. In practice, the `deffixequiv` and `deffixequiv-mutual` utilities usually fully automate the derivation of the congruence from the transparency theorem.

Even if we only need to write a `deffixequiv` or `deffixequiv-mutual` form after each definition, this can be easy to forget. To further automate following the discipline, you can optionally enable a *post-define hook* that will automatically issue a suitable `deffixequiv` command after each definition. See the documentation for `fixequiv-hook` for details.

4 Challenges and Future Work

The FTY library provides a robust implementation of a type system that would feel familiar to users of strongly typed functional programming languages such as Haskell or ML. However, there are a few pitfalls in their practical use, which we discuss below along with potential solutions.

4.1 Generic Functions

The most common problem in working with the fixtype discipline is in the use of generic functions such as `assoc`, `append`, and many others. These functions are designed to work on objects of nonspecific type, and therefore don't follow fixing conventions for specific types. One can always apply appropriate fixing functions to the inputs of these functions, so programming with them in a fixtype discipline isn't hard. However, applying this simple strategy to theorems will often result in ineffective rewrite rules.

For example, suppose `(bind-square-to-root key alist)` fixes `key` to type `natp` and `alist` to type `nat-nat-alist-p`, and we want to prove a theorem like the following:

```
(equal (assoc k (bind-square-to-root k rest))
       (or (and (square-p k)
                (cons k (nat-sqrt k)))
           (assoc k rest)))
```

Presumably this isn't true without some type assumptions. One way to fix the theorem is to apply fixing functions everywhere that typed variables are used in generic contexts:

```
(equal (assoc (nfix k) (bind-square-to-root k rest))
       (or (and (square-p k)
                (cons (nfix k) (nat-sqrt k)))
           (assoc (nfix k) (nat-nat-alist-fix rest))))
```

But this rewrite rule is not always applicable. The left hand side will match only when we have an explicit `nfix` in our goal, but this `nfix` is likely to be simplfied away in cases where the key is known to be a natural. In these cases, a formulation with a type hypothesis would work:

```
(implies (natp k)
         (equal (assoc k (bind-square-to-root k rest))
                ...))
```

Unfortunately, this rule typically won't allow us to simplify terms such as:

```
(assoc (nfix k) (bind-square-to-root k rest))
```

because it fails to unify. In general, both kinds of terms may be encountered and both formulations of the rule may be useful. To solve this problem, one might consider automation to generate both forms of the theorem, or to generate a theorem that catches both cases as follows:

```
(implies (and (syntaxp (or (equal k1 k) (equal k1 `(nfix ,k))))
              (nat-equiv k1 k)
              (natp k1))
         (equal (assoc k1 (bind-square-to-root k rest))
                ...))
```

For the moment, unfortunately, reasoning about a mix of generic functions with fixtype-discipline functions seems to require the sort of consideration of types that we had hoped to avoid. We generally deal with these problems on an ad-hoc basis, either by proving both forms of the theorem or, in more problematic cases, by introducing typed alternatives to the generic functions involved.

4.2 Subtypes

It is possible to use the FTY library to define two types that have a subtype relation, but the library doesn't have any automation for proving or making use of this relationship.

In practice, we have found it difficult to get subtype relations to work well. Proving theorems about a mixture of functions that operate on sub- and supertypes has the same problems as proving theorems with a mixture of generic and fixtype functions, as discussed above. Reasoning about a subtype hierarchy also can lead to degraded prover performance, since proving that something is of type A may lead by backchaining to attempting to prove it to be of each subtype of A.

4.3 Parameterized Types

Haskell and ML support types that take other types as parameters, e.g., List A signifying a list of objects of type A, where A is a type variable. Function signatures may contain types that are not fully specified, and these functions may later be used in contexts where the type variables are concretized as particular types.

Selfridge and Smith [17] created a macro library that supports a form of polymorphism by automating the creation of instances of the `defsum` macro. Polymorphic functions are then supported by another set of macros that allow one to instantiate a template function definition with different substitutions for type variables. A similar macro library could be used to add polymorphism via templates to FTY, but this has not yet been done.

4.4 Dependent Types

Correct behavior of multiple-input functions often depends on constraints involving more than one of the inputs. The fixtypes discipline is focused on unary types, but occasionally it is desirable for a product type to contain multiple elements that have constraints linking them. We have experimentally implemented support for this in `defprod` and `deftagsum` by allowing the user to specify these constraints along with an extra fixing step that forces the fields to satisfy these constraints; this works in practice for simple constraints like "a literal's value should fit into its width." We expect that there would be difficulties in formulating constraints between subfields of a recursive data structure.

4.5 Symbolic and Logical Evaluation

Execution efficiency of functions using the fixtype discipline is highly dependent on the use of `mbe` to avoid calling fixing functions. Evaluation in the logic (with guard checking turned off) is much more expensive with such functions because the `:logic` part of the `mbe` then needs to be executed.

This problem also applies to symbolic evaluation with the GL system [18]. GL is used in hardware verification at Centaur and elsewhere; it evaluates ACL2 functions on bit-level symbolic inputs, producing bit-level symbolic results, allowing the use of SAT or BDD reasoning to prove ACL2 theorems. However, GL ignores guards (except for concrete evaluation) and instead symbolically simulates the logical definitions of functions. Therefore, when using GL on fixtype-compliant functions, these fixing functions will be unnecessarily (symbolically) executed frequently.

This extra expense could be problematic in some cases. In future work we expect to address this by adding a facility to FTY to generate extra GL rules to help it avoid executing fixing functions. Currently, we have worked around this problem in some cases by using fixing functions that are cheap to symbolically execute. For example, if we are dealing with, say, a 32-bit unsigned integer type, then for symbolic simulation it is cheaper to use `(loghead 32 x)` rather than `(if (unsigned-byte-p 32 x) x 0)` as the fixing function. This expense of fixing can also be avoided by creating custom symbolic counterparts, which are used in important core routines in hardware verification frameworks like `ESIM` and `SV`.

4.6 Traversal of Complicated Data Structures

In languages like ML or Haskell, it is possible to write higher order functions for traversing deeply nested data structures. This capability goes a long way toward making it reasonable to inspect and manipulate such objects. Since ACL2 is first order, we cannot write these kinds of generic traversals. Instead, we have to duplicate the boilerplate code for traversing a structure in each algorithm that operates on it. This can become very tedious.

For example, the parsetree format for the VL Verilog/SystemVerilog toolkit contains 168 datatypes, 132 of which are defined in terms of other types (as a product, list, etc.), reflecting the complexity of the SystemVerilog language. We might like to, for instance, collect all identifiers used in a module. We might also like simplify all expressions throughout a module. Doing either of these will require traversing many of the same structures (modules, declarations, assignments, etc.) to reach the objects of interest (identifiers, expressions).

We have implemented an experimental utility, `defvisitor`, intended to generate the boilerplate code necessary for these situations. The user provides code to be run on certain types and to combine results from recursive calls, and the utility generates the boilerplate to traverse the data structures. The current implementation is a proof of concept and its user interface is likely to change, but it has been used to implement several algorithms within the VL library.

5 Related Work

5.1 Fixing Conventions

The use of fixing conventions to avoid hypotheses is well studied and has been used since the earliest Boyer Moore provers. In Boyer and Moore's 1979 *A Computational Logic* we find a `fix` function for NQTHM's naturals and hypothesis-free theorems such as the commutativity of plus. Boyer and Moore

credit A. P. Morse as the inspiriation for this approach, citing his treatment of set theory, *A Theory of Sets* [16], and recalling[1] that:

> "Morse tried every way he could to fix every function that he introduced to eliminate hypotheses if possible, without doing any damage. He delighted in such theorems as that *and* and *or* distributed over one another for all arguments, no matter what objects the arguments were, no matter that *and* was the exact same as set intersection and that *or* was the same as set union."

In their 1994 *Design Goals for ACL2*, Kaufmann and Moore reflect that "NQTHM's logic gets incredible mileage out of the notion that functions—especially arithmetic functions—default 'unexpected' inputs to reasonable values so that many theorems are stated without hypotheses." As a result, fixing conventions were used liberally in their new theorem prover, e.g., throughout its completion axioms for primitive functions on numbers, characters, strings, etc.

Since then, many ACL2 libraries such as `std/osets` [5] and `bitops`, have made heavy use of the technique. Perhaps the most extreme examples are found in `misc/records` [14] and related work such as typed records [11], `defexec`-enhanded records [12], memories [6], and `defrstobj`, which each use sophisticated, convoluted fixing functions to achieve hypothesis-free read-over-write theorems.

Fixing conventions are often unnecessary in typed logics. In such a logic, when we define functions such as PLUS : NAT $\times$ NAT $\rightarrow$ NAT, there is no need to include any type hypotheses in theorems such as the commutativity of PLUS, because any attempt to call or reason about PLUS on non-NAT arguments is simply an error. On the other hand, even in such a logic, fixing conventions may be useful for modeling behavior of operations whose intended domains are not easy to describe using types. Lamport and Paulson [15] provide an engaging discussion of these sorts of issues.

5.2 Data Struture Libraries

There has been significant previous work to develop data structure libraries for ACL2. An early example is Brock's classic `data-structures` library, which featured macros such as `defstructure` [3]. As a concrete example of using this macro, we might write:

```
(defstructure student
  (name (:assert (stringp name) :type-prescription))
  (age  (:assert (natp age) :type-prescription)))
```

This produces a constructor that simply conses together its arguments and accessors that simply `car`/`cdr` into their argument. No fixing is done, so the constructor only produces a well-formed `student-p` if its arguments have the proper types, and the accessors may produce ill-typed results when applied to non-`student-p` objects. Accordingly, reasoning about such structures typically requires type hypotheses. The more recent `defaggregate` macro follows this same approach.

ACL2's single threaded objects [2] are in many ways similar to `defstructure` and `defaggregate`. Although it probably would make little sense to define a student structure as a stobj, we can do so:

```
(defstobj student
  (name :type string        :initially "")
  (age  :type (integer 0 *) :initially 0))
```

[1] Correspondence with Bob Boyer and J Moore.

The resulting recognizer, accessors, and mutators are similar to those produced by `defstructure` or `defaggregate` and so reasoning about these operations usually requires type hypotheses. On the other hand, the recent addition of abstract stobjs [10] makes it possible to develop alternative logical interfaces, e.g., we could arrange so that the `student` stobj was logically viewed as an FTY product object.

The `defdata` [4] library by Chamarthi, Dillinger, and Manolios features an alternative macro, also called `defdata`, that supports introducing richer types, such as sum types, mutually recursive types, etc. This framework also features integrated support for counterexample generation, an exciting feature which FTY does not yet have. To define a similar `student` structure with `defdata` we might write:

```
(defdata student (record (name . string)
                         (age  . nat)))
```

This similarly results in a `studentp` recognizer, constructor, and accessors like `student-age`. Unlike `defstructure` or `defaggregate`, the underlying representation of these functions is based on an optimized variant [12] of Kaufmann and Sumners' records book [14], and the macro provides special integration with ACL2's `Tau` reasoning procedure. However, the general approach to type reasoning about these structures is unchanged: we still require type hypotheses to establish that the construct produces a valid `studentp`, that `student-name` returns a string, and so forth.

Despite type hypotheses, macros like `defstructure`, `defaggregate`, and `defdata` are certainly very useful. `Defstructure` has long been used in ACL2 developments, including recent work such as the modeling by Hardin, et al. [13] of the LLVM compiler project's intermediate form in ACL2, and the formalization by van Gastel and Schmaltz [9] of the xMAS language for communication networks on multi-core processors and systems-on-chip. For many years, we used `defaggregate` and other `std/util` macros at Centaur as the basis for our VL library, microcode model [7], and other internal applications. In our experience, porting these libraries to FTY was not difficult and has helped to simplify our code.

5.3 Make-Event Metaprogramming

In 2004, Vernon Austel developed [1] an experimental variant of ACL2 that added support for a certain type system. He explained that this work had required modifying ACL2 because "a usable type system must constantly extend the set of functions whose type it knows about; this seems to require storing type information in the ACL2 world, which macros currently cannot do," and proposed extending ACL2 with something like `make-event` to "allow others to experiment with type systems without having to hack the system code." Indeed, our FTY library makes extensive use of `make-event` to record the associations between type recognizers, fixing functions, and equivalence relations, and to look up (via `define`) the type signatures for functions.

6 Conclusion

FTY is a new data structure library for ACL2 that provides deep support for using fixing functions to avoid type hypotheses in theorems. Its successful use may require somewhat more discipline than similar libraries such as `std/util` or `defdata`. In exchange, it provides a strongly typed programming environment that can help to catch errors during development while largely avoiding type hypotheses during theorem proving.

Having a good data structure library is tremendously useful when developing large systems in ACL2. A fixing discipline is one part of this, but FTY is also increasingly mature and capable in other ways,

e.g., it retains much of the `std/util` look and feel, with features such as `XDOC` integration, convenient `b*` binders, readable `make/change` macros, etc. We are now using FTY as for large ACL2 libraries such as SV and VL libraries, and have been pleased with the results.

The source code for FTY is included in the ACL2 Community Books under the `centaur/fty` directory. Beyond this paper, the FTY library has extensive documentation, which includes more detailed information on the available options for each macro. The `centaur/fty` directory also includes various test cases that may serve as useful examples of using the library.

We hope you find FTY useful.

6.1 Acknowledgments

We thank Bob Boyer and J Moore for very interesting discussions about the origins of fixing disciplines in Boyer-Moore provers. We thank Shilpi Goel and Cuong Chau for their corrections and feedback on this paper.

References

[1] Vernon Austel (2004): *Adding a typing mechanism to ACL2*. ACL2 '04. Available at `http://www.cs.utexas.edu/users/moore/acl2/workshop-2004/contrib/austel/acl2-types.pdf`.

[2] Robert S. Boyer & J Strother Moore (2002): *Single-Threaded Objects in ACL2*. In: *Practical Aspects of Declarative Languages*, *LNCS* 2257, Springer, pp. 9–27, doi:10.1007/3-540-45587-6_3.

[3] Bishop Brock (1997): *Defstructure for ACL2*. Available at `http://www.cs.utexas.edu/users/moore/publications/others/defstructure-brock.ps`.

[4] Harsh Raju Chamarthi, Peter C. Dillinger & Panagiotis Manolios (2014): *Data Definitions in the ACL2 Sedan*. In: *ACL2 '14*, EPTCS, pp. 27–48, doi:10.4204/EPTCS.152.3.

[5] Jared Davis (2004): *Finite Set Theory based on Fully Ordered Lists*. ACL2 '04. Available at `http://www.cs.utexas.edu/users/moore/acl2/workshop-2004/contrib/davis/set-theory.pdf`.

[6] Jared Davis (2006): *Memories: Array-like records for ACL2*. In: *ACL2 '06*, ACM, pp. 57–60, doi:10.1145/1217975.1217986.

[7] Jared Davis, Anna Slobodova & Sol Swords (2014): *Microcode Verification: Another Piece of the Microprocessor Verification Puzzle*. In: *ITP '14*, *LNCS* 8558, Springer, pp. 1–16, doi:10.1007/978-3-319-08970-6_1.

[8] Jared Davis & Sol Swords (2015): *XDOC Documentation for FTY*. Available at `http://www.cs.utexas.edu/users/moore/acl2/manuals/`.

[9] Bernard van Gastel & Julien Schmaltz (2013): *A formalisation of XMAS*. In: *ACL2 '13*, EPTCS, pp. 111–126, doi:10.4204/EPTCS.114.9.

[10] Shilpi Goel, Warren A. Hunt, Jr. & Matt Kaufmann (2013): *Abstract Stobjs and their application to ISA modeling*. In: *ACL2 '13*, EPTCS, pp. 54–69, doi:10.4204/EPTCS.114.5.

[11] David Greve & Matthew Wilding (2003): *Typed ACL2 Records*. ACL2 '03. Available at `http://www.cs.utexas.edu/users/moore/acl2/workshop-2003/contrib/greve-wilding_defrecord/defrecord.pdf`.

[12] David A. Greve, Matt Kaufmann, Panagiotis Manolios, J Strother Moore, Sandip Ray, José Luis Ruiz-Reina, Rob Sumners, Daron Vroon & Matthew Wilding (2008): *Efficient execution in an automated reasoning environment*. *Journal of Functional Programming* 18, pp. 15–46, doi:10.1017/S0956796807006338.

[13] David S. Hardin, Jennifer A. Davis, David A. Greve & Jedidiah R. McClurg (2014): *Development of a Translator from LLVM to ACL2*. In: *ACL2 '14*, EPTCS, pp. 163–177, doi:10.4204/EPTCS.152.13.

[14] Matt Kaufmann & Rob Sumners (2002): *Efficient Rewriting of Data Structures in ACL2*. ACL2 '02. Available at `http://www.cs.utexas.edu/users/moore/acl2/workshop-2002/contrib/kaufmann-sumners/rcd.pdf`.

[15] Leslie Lamport & Lawrence C. Paulson (1999): *Should Your Specification Language Be Typed*. *ACM Transactions on Programming Languages and Systems* 21(3), pp. 502–526, doi:10.1145/319301.319317.

[16] Anthony P. Morse (1965): *A Theory of Sets*. Academic Press.

[17] Benjamin Selfridge & Eric Smith (2014): *Polymorphic Types in ACL2*. In: *ACL2 '14*, EPTCS, pp. 49–60, doi:10.4204/EPTCS.152.4.

[18] Sol Swords & Jared Davis (2011): *Bit-Blasting ACL2 Theorems*. In: *ACL2 '11*, *Electronic Proceedings in Theoretical Computer Science* 70, pp. 84–102, doi:10.4204/EPTCS.70.7.

Second-Order Functions and Theorems in ACL2

Alessandro Coglio
Kestrel Institute
http://www.kestrel.edu/~coglio

SOFT ('Second-Order Functions and Theorems') is a tool to mimic second-order functions and theorems in the first-order logic of ACL2. Second-order functions are mimicked by first-order functions that reference explicitly designated uninterpreted functions that mimic function variables. First-order theorems over these second-order functions mimic second-order theorems universally quantified over function variables. Instances of second-order functions and theorems are systematically generated by replacing function variables with functions. SOFT can be used to carry out program refinement inside ACL2, by constructing a sequence of increasingly stronger second-order predicates over one or more target functions: the sequence starts with a predicate that specifies requirements for the target functions, and ends with a predicate that provides executable definitions for the target functions.

1 The SOFT Tool

SOFT ('Second-Order Functions and Theorems') is a tool to mimic second-order functions and theorems [4] in the first-order logic of ACL2 [3]. Second-order functions are mimicked by first-order functions that reference explicitly designated uninterpreted functions that mimic function variables. First-order theorems over these second-order functions mimic second-order theorems universally quantified over function variables. Instances of second-order functions and theorems are systematically generated by replacing function variables with functions. Theorem instances are proved automatically, via automatically generated functional instantiations [5].

SOFT does not extend the ACL2 logic. It is an ACL2 library, available in the ACL2 community books, that provides macros to introduce function variables, second-order functions, second-order theorems, and instances thereof. The macros modify the ACL2 state only by submitting sound and conservative events; they cannot introduce unsoundness or inconsistency on their own. The main features of the macros are described and exemplified below; full details are in their documentation and implementation.

1.1 Function Variables

A *function variable* is introduced as

```
(defunvar fv (* ... *) => *)
```

where:

- `fv` is a symbol, which names the function variable.
- `(* ... *)` is a list of 1 or more `*`s, which defines the arity, i.e. type [6], of `fv`.

This generates the event

```
(defstub fv (* ... *) => *)
```

i.e. `fv` is introduced as an uninterpreted function with the given type. Furthermore, a `table` event is generated to record `fv` in a global table of function variables.

For example,

M. Kaufmann and D. Rager (Eds.): ACL2 Workshop 2015 (ACL2 2015).
EPTCS 192, 2015, pp. 17–33, doi:10.4204/EPTCS.192.3

```
(defunvar ?f (*) => *)
(defunvar ?p (*) => *)
(defunvar ?g (* *) => *)
```

introduce two unary function variables and one binary function variable. Starting function variable names with `?` provides a visual cue for their function variable status, but SOFT does not enforce this naming convention.

1.2 Second-Order Functions

SOFT supports three kinds of second-order functions: plain second-order functions, choice second-order functions, and quantifier second-order functions.

1.2.1 Plain Functions

A *plain second-order function* is introduced as

`(defun2` *sof* `(`fv_1 `...` fv_n`) (`v_1 `...` v_m`)` *doc decl ... decl body*`)`

where:

- *sof* is a symbol, which names the second-order function.
- `(`fv_1 `...` fv_n`)` is a non-empty list without duplicates of previously introduced function variables, whose order is immaterial, which are the function parameters of *sof*.
- The other items are as in `defun`: individual variables, optional documentation string, optional declarations, and defining body.
- $\mathsf{FV}(\mathit{body}) \cup \mathsf{FV}(\mathit{measure}) \cup \mathsf{FV}(\mathit{guard}) = \{fv_1, \ldots, fv_n\}$, where:
  - *measure* is the measure expression of *sof*, or `nil` if *sof* is not recursive.
  - *guard* is the guard of *sof* (`t` if not given explicitly in the declarations).
  - $\mathsf{FV}(\mathit{term})$ is the set of function variables that either occur in *term* or are function parameters of second-order functions that occur in *term*.

  I.e. the function parameters of *sof* are all and only the function variables that *sof* depends on.[1]

This generates the event

`(defun` *sof* `(`v_1 `...` v_m`)` *doc decl ... decl body*`)`

i.e. *sof* is introduced as a first-order function using `defun`, removing the function variables. Furthermore, a `table` event is generated to record *sof* in a global table of second-order functions.

For example,

```
(defun2 quad[?f] (?f) (x)
  (?f (?f (?f (?f x)))))
```

introduces a non-recursive function to apply its function parameter to its individual parameter four times. The name `quad[?f]` conveys the dependency on the function parameter and provides a visual cue for the implicit presence of the function parameter when the function is applied, e.g. in `(quad[?f] x)`, but SOFT does not enforce this naming convention.

As another example,

[1] Thus, `defun2` could have been defined to have the same form as `defun`, i.e. without `(`fv_1 `...` fv_n`)`. However, the presence of the functions parameters parallels that of the individual parameters, and the redundancy check may detect user errors.

```
(defun2 all[?p] (?p) (l)
  (cond ((atom l) (null l))
        (t (and (?p (car l)) (all[?p] (cdr l))))))
```

introduces a recursive predicate (i.e. boolean-valued function) that recognizes `nil`-terminated lists whose elements satisfy the predicate parameter.

As a third example,

```
(defun2 map[?f_?p] (?f ?p) (l)
  (declare (xargs :guard (all[?p] l)))
  (cond ((endp l) nil)
        (t (cons (?f (car l)) (map[?f_?p] (cdr l))))))
```

introduces a recursive function that homomorphically lifts `?f` to operate on `nil`-terminated lists whose elements satisfy `?p`. The predicate parameter `?p` only appears in the guard, not in the body.

As a fourth example,

```
(defun2 fold[?f_?g] (?f ?g) (bt)
  (cond ((atom bt) (?f bt))
        (t (?g (fold[?f_?g] (car bt)) (fold[?f_?g] (cdr bt))))))
```

introduces a generic folding function on values as binary trees.

1.2.2 Choice Functions

A *choice second-order function* is introduced as

`(defchoose2` *sof* `(`bv_1 `...` bv_p`) (`fv_1 `...` fv_n`) (`v_1 `...` v_m`)` *body key-opts*`)`

where:

- *sof* is a symbol, which names the second-order function.
- `(`fv_1 `...` fv_n`)` are the function parameters, as in `defun2`.
- The other items are as in `defchoose`: bound variables, individual variables, constraining body, and keyed options.
- $\mathsf{FV}(body) = \{fv_1, \ldots, fv_n\}$.

This generates the event

`(defchoose` *sof* `(`bv_1 `...` bv_p`) (`v_1 `...` v_m`)` *body key-opts*`)`

i.e. *sof* is introduced as a first-order function using `defchoose`, removing the function variables. Furthermore, a `table` event is generated to record *sof* in the same global table where plain second-order functions are recorded.

For example,

```
(defchoose2 fixpoint[?f] x (?f) ()
  (equal (?f x) x))
```

introduces a second-order function constrained to return a fixed point of `?f`, if any exists.

1.2.3 Quantifier Functions

A *quantifier second-order function* is introduced as

`(defun-sk2` *sof* `(`fv_1 `...` fv_n`) (`v_1 `...` v_m`)` *body key-opts*`)`

where:

- *sof* is a symbol, which names the second-order function.
- `(`fv_1 `...` fv_n`)` are the function parameters, as in `defun2` and `defchoose2`.
- The other items are as in `defun-sk`: individual variables, defining body, and keyed options.
- $\mathsf{FV}(\mathit{body}) \cup \mathsf{FV}(\mathit{guard}) = \{fv_1, \ldots, fv_n\}$, where *guard* is the guard of *sof* (`t` if not given explicitly in the `:witness-dcls` option).

This generates the event

`(defun-sk` *sof* `(`v_1 `...` v_m`)` *body key-opts*`)`

i.e. *sof* is introduced as a first-order function using `defun-sk`, removing the function variables. Furthermore, a `table` event is generated to record *sof* in the same global table where plain and choice second-order functions are recorded.

For example,

```
(defun-sk2 injective[?f] (?f) ()
  (forall (x y) (implies (equal (?f x) (?f y)) (equal x y))))
```

introduces a predicate that recognizes injective functions.

1.3 Instances of Second-Order Functions

An *instance of a second-order function* is a function introduced as

`(defun-inst` *f* `(`fv_1 `...` fv_n`) (`*sof* `(`fv_1`' . `f_1`') ... (`fv_m`' . `f_m`')) `*key-opts*`)`

where:

- *f* is a symbol, which names the new function.
- `(`fv_1 `...` fv_n`)` are optional function parameters. If present, *f* is a second-order function; if absent, *f* is a first-order function.
- *sof* is a previously introduced second-order function.
- `((`fv_1`' . `f_1`') ... (`fv_m`' . `f_m`'))` is an *instantiation* Σ, i.e. an alist whose keys fv_i' are distinct function variables, whose values f_i' are previously introduced function variables, second-order functions, or regular first-order functions, and where each f_i' has the same type as fv_i'. Each fv_i' is a function parameter of *sof*. The notation `(`*sof* `(`fv_1`' . `f_1`') ... (`fv_m`' . `f_m`'))` suggests the application of *sof* to the functions f_i'; since the function parameters of *sof* are unordered, the application is by explicit association, not positional. An instance of a second-order function is introduced as a named application of the second-order function; SOFT does not support the application of a second-order function on the fly within a term, as in the application of a first-order function. Not all the function parameters of *sof* must be keys in Σ; missing function parameters are left unchanged.
- *key-opts* are keyed options, e.g. to override attributes of *f* that are otherwise derived from *sof*.

- If *sof* is a plain function, $\mathsf{FV}(\Sigma(\mathit{body})) \cup \mathsf{FV}(\Sigma(\mathit{measure})) \cup \mathsf{FV}(\Sigma(\mathit{guard})) = \{\mathit{fv}_1, \ldots, \mathit{fv}_n\}$, where *body*, *measure*, and *guard* are the body, measure expression (`nil` if *sof* is not recursive), and guard of *sof*, and $\Sigma(\mathit{term})$ is the result of applying Σ to *term* (see below).
- If *sof* is a choice function, $\mathsf{FV}(\Sigma(\mathit{body})) = \{\mathit{fv}_1, \ldots, \mathit{fv}_n\}$, where *body* is the body of *sof*.
- If *sof* is a quantifier function, $\mathsf{FV}(\Sigma(\mathit{body})) \cup \mathsf{FV}(\Sigma(\mathit{guard})) = \{\mathit{fv}_1, \ldots, \mathit{fv}_n\}$, where *body* and *guard* are the body and guard of *sof*.

This generates a `defun`, `defchoose`, or `defun-sk` event, depending on whether *sof* is a plain, choice, or quantifier function. The event introduces *f* with body $\Sigma(\mathit{body})$, measure $\Sigma(\mathit{measure})$ (if *sof* is recursive, hence plain), and guard $\Sigma(\mathit{guard})$ (if *sof* is a plain or quantifier function). *f* is recursive iff *sof* is recursive: `defun-inst` generates the termination proof of *f* from the termination proof of *sof* using the techniques to instantiate second-order theorems described in Section 1.5.

Furthermore, `defun-inst` generates a `table` event to record *f* as the Σ instance of *sof* in a global table of instances of second-order functions. If *f* is second-order, `defun-inst` also generates a `table` event to record *f* in the global table of second-order functions.

$\Sigma(\mathit{term})$ is obtained from *term* by replacing the keys of Σ in *term* with their values in Σ. This involves not only explicit occurrences of such keys in *term*, but also implicit occurrences as function parameters of second-order functions occurring in *term*. For example, if the pair `(?f . f)` is in Σ, `sof[...?f...]` is a second-order function whose function parameters include `?f`, and *term* is `(cons (?f x) (sof[...?f...] y))`, then $\Sigma(\mathit{term})$ is `(cons (f x) (sof[...f...] y))`, where `sof[...f...]` is the Σ' instance of `sof[...?f...]`, where Σ' is the restriction of Σ to the keys that are function parameters of `sof[...?f...]`. The table of instances of second-order functions is consulted to find `sof[...f...]`. If the instance is not in the table, `defun-inst` fails: the user must introduce `sof[...f...]`, via a `defun-inst`, and then re-try the failed instantiation.

For example, given a function

```
(defun wrap (x) (list x))
```

that wraps a value into a singleton list,

```
(defun-inst quad[wrap]
  (quad[?f] (?f . wrap)))
```

introduces a function that wraps a value four times.

As another example, given a predicate

```
(defun octetp (x) (and (natp x) (< x 256)))
```

that recognizes octets,

```
(defun-inst all[octetp]
  (all[?p] (?p . octetp)))
```

introduces a predicate that recognizes `nil`-terminated lists of octets.

As a third example,

```
(defun-inst map[code-char]
  (map[?f_?p] (?f . code-char) (?p . octetp)))
```

introduces a function that translates lists of octets to lists of corresponding characters. The replacement `code-char` of `?f` induces the replacement `octetp` of `?p`, because the guard of `code-char` is (equivalent to) `octetp`; the name `map[code-char]` indicates only the replacement of `?f` explicitly.

As a fourth example,

```
(defun-inst fold[nfix_plus]
  (fold[?f_?g] (?f . nfix) (?g . binary-+)))
```

adds up all the natural numbers in a tree, coercing other values to 0.

As a fifth example, given a function

```
(defun twice (x) (* 2 (fix x)))
```

that doubles a value,

```
(defun-inst fixpoint[twice]
  (fixpoint[?f] (?f . twice)))
```

introduces a function constrained to return the (only) fixed point 0 of `twice`.

As a sixth example,

```
(defun-inst injective[quad[?f]] (?f)
  (injective[?f] (?f . quad[?f])))
```

introduces a predicate that recognizes functions whose four-fold application is injective.

1.4 Second-Order Theorems

A *second-order theorem* is a theorem whose formula depends on function variables, which occur in the theorem or are function parameters of second-order functions that occur in the theorem. Since function variables are unconstrained, a second-order theorem is effectively universally quantified over the function variables that it depends on. It is introduced via standard events like `defthm`.[2]

For example,

```
(defthm len-of-map[?f_?p]
  (equal (len (map[?f_?p] l)) (len l)))
```

shows that the homomorphic lifting of `?f` to lists of `?p` values preserves the length of the list, for every function `?f` and predicate `?p`.

As another example,

```
(defthm injective[quad[?f]]-when-injective[?f]
  (implies (injective[?f]) (injective[quad[?f]]))
  :hints
  (("Goal" :use
    ((:instance
      injective[?f]-necc
      (x (?f (?f (?f (?f (mv-nth 0 (injective[quad[?f]]-witness)))))))
      (y (?f (?f (?f (?f (mv-nth 1 (injective[quad[?f]]-witness))))))))
     (:instance
      injective[?f]-necc
      (x (?f (?f (?f (mv-nth 0 (injective[quad[?f]]-witness))))))
      (y (?f (?f (?f (mv-nth 1 (injective[quad[?f]]-witness)))))))
     (:instance
      injective[?f]-necc
```

[2]The absence of an explicit quantification over function variables in second-order theorems parallels the absence of an explicit quantification over individual variables in first-order theorems.

```
 (x (?f (?f (mv-nth 0 (injective[quad[?f]]-witness)))))
 (y (?f (?f (mv-nth 1 (injective[quad[?f]]-witness))))))
(:instance
 injective[?f]-necc
 (x (?f (mv-nth 0 (injective[quad[?f]]-witness))))
 (y (?f (mv-nth 1 (injective[quad[?f]]-witness)))))
(:instance
 injective[?f]-necc
 (x (mv-nth 0 (injective[quad[?f]]-witness)))
 (y (mv-nth 1 (injective[quad[?f]]-witness)))))))
```

shows that the four-fold application of an injective function is injective.

As a third example, given a function variable

```
(defunvar ?io (* *) => *)
```

for an abstract input/output relation, a predicate

```
(defun-sk2 atom-io[?f_?io] (?f ?io) ()
  (forall x (implies (atom x) (?io x (?f x))))
  :rewrite :direct)
```

that recognizes functions `?f` that satisfy the input/output relation on atoms, and a predicate

```
(defun-sk2 consp-io[?g_?io] (?g ?io) ()
  (forall (x y1 y2)
          (implies (and (consp x) (?io (car x) y1) (?io (cdr x) y2))
                   (?io x (?g y1 y2))))
  :rewrite :direct)
```

that recognizes functions `?g` that satisfy the input/output relation on `cons` pairs when the arguments are valid outputs for the `car` and `cdr` components,

```
(defthm fold-io[?f_?g_?io]
  (implies (and (atom-io[?f_?io]) (consp-io[?g_?io]))
           (?io x (fold[?f_?g] x))))
```

shows that the generic folding function on binary trees satisfies the input/output relation when its function parameters satisfy the predicates just introduced.

1.5 Instances of Second-Order Theorems

An *instance of a second-order theorem* is a theorem introduced as

`(defthm-inst` *thm* `(`*sothm* `(`fv_1 `.` f_1`) ... (`fv_n `.` f_n`)) :rule-classes ...)`

where:

- *thm* is a symbol, which names the new theorem.
- *sothm* is a previously introduced second-order theorem.
- `((`fv_1 `.` f_1`) ... (`fv_n `.` f_n`))` is an instantiation Σ, where each fv_i is a function variable that *sothm* depends on. The notation `(`*sothm* `(`fv_1 `.` f_1`) ... (`fv_m `.` f_m`))` is similar to `defun-inst`.
- The keyed option `:rule-classes ...` is as in `defthm`.

This generates the event

(defthm *thm* Σ(*formula*) :rule-classes ... :instructions *proof*)

where:

- *formula* is the formula of *sothm*.
- *proof* consists of two commands for the ACL2 proof checker to prove *thm* using *sothm*.

The first command of *proof* is

(:use (:functional-instance *sothm* (*fv*$_1$ *f*$_1$) ... (*fv*$_n$ *f*$_n$) *more-pairs*))

i.e. *thm* is proved using a functional instance of *sothm*. The pairs that define the functional instance include not only the pairs that form Σ (in list notation instead of dotted notation), but also, in *more-pairs* above, all the pairs (*sof f*) such that *sof* is a second-order function that occurs in *sothm* and *f* is its replacement in *thm* (i.e. *f* is the Σ' instance of *sof*, where Σ' is the restriction of Σ to the function parameters of *sof*). These additional pairs are determined in the same way as when Σ is applied to *formula* (see Section 1.3): thus, the result of (:functional-instance ...) above is Σ(*formula*), and the main goal of *thm* is readily proved.

The use of the functional instance reduces the proof of *thm* to proving that, for each pair, the replacing function satisfies all the constraints of the replaced function. Since function variables are unconstrained, nothing needs to be proved for the (*fv*$_i$ *f*$_i$) pairs. For each (*sof f*) pair in *more-pairs*, it must be proved that *f* satisfies the constraints on *sof*. If *sof* references another second-order function *sof'* that depends on some *fv*$_i$, a further pair (*sof' f'*) goes into *more-pairs*, where *f'* is the appropriate instance of *sof'*, so that the constraints on *sof* to be proved are properly instantiated. This further pair generates further constraints to be proved. To properly instantiate these further constraints, another pair (*sof'' f''*) goes into *more-pairs*, if *sof''* is a second-order function referenced by *sof'* that depends on some *fv*$_i$, and *f''* is the appropriate instance of *sof''*. Therefore, *more-pairs* includes all the pairs (*sof f*) such that *sof* is a second-order function that is directly or indirectly referenced by *sothm* and that depends on some *fv*$_i$, and *f* is the appropriate instance of *sof*.

If *sof* is a quantifier second-order function, it references a witness function *sof*$_w$ introduced by defun-sk. The defun-sk that introduces the instance *f* of *sof* also introduces a witness function *f*$_w$ that is effectively an instance of *sof*$_w$, but is not recorded in the table of instances of second-order functions because *sof*$_w$ and *f*$_w$ are "internal". The pair (*sof*$_w$ *f*$_w$) goes into *more-pairs* as well.

For each pair (*sof f*) in *more-pairs*, the constraints of *sof* are: the definition of *sof* if *sof* is a plain function; the constraining axiom of *sof* if *sof* is a choice function; the definition of *sof* and the rewrite rule of *sof* if *sof* is a quantifier function (the rewrite rule of *sof* is generated by defun-sk; its default name is *sof*-necc if the quantifier is universal, *sof*-suff if the quantifier is existential). Instantiating these constraints yields the corresponding definitions, constraining axioms, and rewrite rules of *f*, by the construction of the instance *f* of *sof*.

The second command of *proof* is

(:repeat (:then (:use *facts*) :prove))

where *facts* includes the names of all the *f* functions in *more-pairs*, which are also the names of their definitions and constraining axioms; *facts* also includes the names of the rewrite rules for quantifier functions. This command runs the prover on every proof subgoal, after augmenting each subgoal with all the facts in *facts*. This command has worked on all the examples tried so far, but a more honed approach could be investigated, should some future example fail; since the constraints are satisfied by construction, this is just an implementation issue.

For example,

```
(defthm-inst len-of-map[code-char]
  (len-of-map[?f_?p] (?f . code-char) (?p . octetp)))
```

shows that `map[code-char]` preserves the length of the list.

As another example, given instances

```
(defun-inst injective[quad[wrap]] (injective[quad[?f]] (?f . wrap)))
(defun-inst injective[wrap] (injective[?f] (?f . wrap)))
```

the theorem instance

```
(defthm-inst injective[quad[wrap]]-when-injective[wrap]
  (injective[quad[?f]]-when-injective[?f] (?f . wrap)))
```

shows that `quad[wrap]` is injective if `wrap` is.

An example instance of `fold-io[?f_?g_?io]` is in Section 2.

1.6 Summary of the Macros

`defunvar`, `defun2`, `defchoose2`, and `defun-sk2` are wrappers of existing events that explicate function variable dependencies and record additional information. They set the stage for `defun-inst` and `defthm-inst`.

`defun-inst` provides the ability to concisely generate functions, and automatically prove their termination if recursive, by specifying replacements of function variables.

`defthm-inst` provides the ability to concisely generate and automatically prove theorems, by specifying replacements of function variables.

2 Use in Program Refinement

In program refinement [9], a correct-by-construction implementation is derived from a requirements specification via a sequence of intermediate specifications. *Shallow pop-refinement* (where 'pop' stands for 'predicates over programs') is an approach to program refinement, carried out inside an interactive theorem prover by constructing a sequence of increasingly stronger predicates over one or more target functions. The sequence starts with a predicate that specifies requirements for the target functions, and ends with a predicate that provides executable definitions for the target functions. Shallow pop-refinement is a form of pop-refinement [8] in which the programs predicated upon are shallowly embedded functions of the logic of the theorem prover, instead of deeply embedded programs of a programming language as in [8].

SOFT can be used to carry out shallow pop-refinement in ACL2, as explained and exemplified below. The example derivation is overkill for the simple program obtained, which can be easily written and proved correct directly. But the purpose of the example is to illustrate techniques that can be used to derive more complex programs, and how SOFT supports these techniques (which are more directly supported in higher-order logic). The hints in some of the theorems below distill their proofs into patterns that should apply to similarly structured derivations, suggesting opportunities for future automation.

2.1 Specifications as Second-Order Predicates

Requirements over $n \geq 1$ target functions are specified by introducing function variables $fv_1, \ldots, fv_n$ that represent the target functions, and by defining a second-order predicate $spec_0$ over $fv_1, \ldots, fv_n$ that

asserts the required properties of the target functions. The possible implementations are all the n-tuples of executable functions that satisfy the predicate. The task is to find such an n-tuple, thus constructively proving the predicate, existentially quantified over the function parameters.

For example, given a function

```
(defun leaf (e bt)
  (cond ((atom bt) (equal e bt))
        (t (or (leaf e (car bt)) (leaf e (cdr bt))))))
```

to test whether something is a leaf of a binary tree, a function to extract from a binary tree the leaves that are natural numbers, in no particular order and possibly with duplicates, can be specified as

```
(defunvar ?h (*) => *)
(defun-sk io (x y) ; input/output relation
  (forall e (iff (member e y) (and (leaf e x) (natp e))))
  :rewrite :direct)
(defun-sk2 spec[?h] (?h) ()
  (forall x (io x (?h x)))
  :rewrite :direct)
```

The task is to solve `spec[?h]` for `?h`, i.e. to find an executable function `h` such that the instance `spec[h]` of `spec[?h]` holds.

Properties implied by the requirements are proved as second-order theorems with $spec_0$ as hypothesis, e.g. for validation purposes. Since the function parameters are universally quantified in the theorem, the properties hold for all the implementations of the specification.

For example, the members of the output of every implementation of `spec[?h]` are natural numbers:

```
(defthm natp-of-member-of-output
  (implies (and (spec[?h]) (member e (?h x))) (natp e))
  :hints (("Goal" :use (spec[?h]-necc (:instance io-necc (y (?h x)))))))
```

2.2 Refinement as Second-Order Predicate Strengthening

The specification $spec_0$ is stepwise refined by constructing a sequence $spec_1, \ldots, spec_m$ of increasingly stronger predicates over $fv_1, \ldots, fv_n$. Each such predicate embodies a decision that either narrows down the possible implementations or rephrases their description towards their determination. The correctness of each step $j \in \{1, \ldots, m\}$ is expressed by the second-order theorem `(implies (`$spec_j$`) (`$spec_{j-1}$`))`.

The sequence ends with $spec_m$ asserting that each fv_i is equal to some executable function f_i:[3]

```
(defun-sk2 def1 (fv1) () (forall x (equal (fv1 x) (f1 x))))
...
(defun-sk2 defn (fvn) () (forall x (equal (fvn x) (fn x))))
(defun2 specm (fv1 ... fvn) () (and (def1) ... (defn)))
```

The tuple $\langle f_1, \ldots, f_n \rangle$ is the implementation. Chaining the implications of the m step correctness theorems yields the second-order theorem `(implies (`$spec_m$`) (`$spec_0$`))`. Its Σ instance, where Σ is the instantiation `((`fv_1 `.` f_1`) ... (`fv_n `.` f_n`))`, is essentially $\Sigma($`(`$spec_0$`)`$)$ (because $\Sigma($`(`$spec_m$`)`$)$ is trivially true), which asserts that the implementation $\langle f_1, \ldots, f_n \rangle$ satisfies $spec_0$.

More precisely, in the course of the derivation, function variables $fv_{n+1}, \ldots, fv_{n+p}$ may be added to represent additional target functions $f_{n+1}, \ldots, f_{n+p}$ called by $f_1, \ldots, f_n$. This may happen as the task

[3] The body of each `(defun-sk2 def`$_i$ `...)` is a first-order expression of the second-order equality $fv_i = f_i$.

of finding $f_1, \ldots, f_n$ is progressively reduced to simpler sub-tasks of finding $f_{n+1}, \ldots, f_{n+p}$. If fv_{n+k} is added at refinement step j, since $spec_{j-1}$ does not depend on fv_{n+k}, the universal quantification of fv_{n+k} over the step correctness theorem `(implies (`$spec_j$`) (`$spec_{j-1}$`))` is equivalent to an existential quantification of fv_{n+k} over the hypothesis `(`$spec_j$`)` of the theorem. The complete implementation that results from the derivation is $\langle f_1, \ldots, f_n, f_{n+1}, \ldots, f_{n+p} \rangle$.

The function variables fv_i are placeholders for the target functions in the $spec_j$ predicates. Each fv_i remains uninterpreted throughout the derivation; no constraints are attached to it via axioms. Each $spec_j$ is defined, so it does not introduce logical inconsistency. Inconsistent requirements on the target functions amount to $spec_0$ being always false, not to logical inconsistency. Obtaining an implementation witnesses the consistency of the requirements.

For example, `spec[?h]` from Section 2.1 can be refined as follows.

Step 1 Since the target function represented by `?h` operates on binary trees, `spec[?h]` is strengthened by constraining `?h` to be the folding function on binary trees from Section 1.2.1:

```
(defun-sk2 def-?h-fold[?f_?g] (?h ?f ?g) ()
  (forall x (equal (?h x) (fold[?f_?g] x)))
  :rewrite :direct)
(defun2 spec1[?h_?f_?g] (?h ?f ?g) ()
  (and (def-?h-fold[?f_?g]) (spec[?h])))
(defthm step1 (implies (spec1[?h_?f_?g]) (spec[?h]))
  :hints (("Goal" :in-theory '(spec1[?h_?f_?g]))))
```

The predicate `spec1[?h_?f_?g]` adds to `spec[?h]` the conjunct `def-?h-fold[?f_?g]`. Thus, the task of finding a solution for `?h` is reduced to the task of finding solutions for `?f` and `?g`: instantiating `def-?h-fold[?f_?g]` with solutions for `?f` and `?g` yields a solution for `?h`, in Step 5 below.

Step 2 The theorem `fold-io[?f_?g_?io]` from Section 1.4, which shows the correctness of the folding function (with respect to an input/output relation) under suitable correctness assumptions on the function parameters, is instantiated with the input/output relation `io` used in `spec[?h]`:

```
(defun-inst atom-io[?f] (?f) (atom-io[?f_?io] (?io . io)))
(defun-inst consp-io[?g] (?g) (consp-io[?g_?io] (?io . io)))
(defthm-inst fold-io[?f_?g] (fold-io[?f_?g_?io] (?io . io)))
```

Since the conclusion `(io x (fold[?f_?g] x))` of `fold-io[?f_?g]` equals the matrix `(io x (?h x))` of `spec[?h]` when `def-?h-fold[?f_?g]` holds, `spec1[?h_?f_?g]` is strengthened by replacing the `spec[?h]` conjunct with the hypotheses of `fold-io[?f_?g]`:

```
(defun2 spec2[?h_?f_?g] (?h ?f ?g) ()
  (and (def-?h-fold[?f_?g]) (atom-io[?f]) (consp-io[?g])))
(defthm step2 (implies (spec2[?h_?f_?g]) (spec1[?h_?f_?g]))
  :hints (("Goal" :in-theory '(spec1[?h_?f_?g] spec2[?h_?f_?g] spec[?h]
                               def-?h-fold[?f_?g]-necc fold-io[?f_?g]))))
```

Step 3 The predicate `atom-io[?f]` specifies requirements on `?f` independently from `?g` and `?h`. An implementation `f` can be derived by constructing a sequence of increasingly stronger predicates over `?f`, in the same way in which `spec[?h]` is being refined stepwise. This is a possible final result:

```
(defun f (x) (if (natp x) (list x) nil))
(defun-inst atom-io[f] (atom-io[?f] (?f . f)))
(defthm atom-io[f]! (atom-io[f]))
```

The predicate `spec2[?h_?f_?g]` is strengthened by replacing the `atom-io[?f]` conjunct with one that constrains `?f` to be `f`:

```
(defun-sk2 def-?f (?f) () (forall x (equal (?f x) (f x))) :rewrite :direct)
(defun2 spec3[?h_?f_?g] (?h ?f ?g) ()
  (and (def-?h-fold[?f_?g]) (def-?f) (consp-io[?g])))
(defthm step3-lemma (implies (def-?f) (atom-io[?f]))
  :hints (("Goal" :in-theory '(atom-io[?f] atom-io[f]-necc
                               atom-io[f]! def-?f-necc))))
(defthm step3 (implies (spec3[?h_?f_?g]) (spec2[?h_?f_?g]))
  :hints (("Goal" :in-theory '(spec2[?h_?f_?g] spec3[?h_?f_?g] step3-lemma))))
```

Step 4 The predicate `consp-io[?g]` specifies requirements on `?g` independently from `?f` and `?h`. An implementation g can be derived by constructing a sequence of increasingly stronger predicates over `?g`, in the same way in which `spec[?h]` is being refined stepwise. This is a possible final result:

```
(defun g (y1 y2) (append y1 y2))
(defun-inst consp-io[g] (consp-io[?g] (?g . g)))
(defthm member-of-append ; used to prove CONSP-IO[G]-LEMMA below
  (iff (member e (append y1 y2)) (or (member e y1) (member e y2))))
(defthm consp-io[g]-lemma ; used to prove CONSP-IO[G]! below
  (implies (and (consp x) (io (car x) y1) (io (cdr x) y2))
           (io x (g y1 y2)))
  :hints (("Goal" :in-theory (disable io) :expand (io x (append y1 y2)))))
(defthm consp-io[g]! (consp-io[g]) :hints (("Goal" :in-theory (disable g))))
```

The predicate `spec3[?h_?f_?g]` is strengthened by replacing the `consp-io[?f]` conjunct with one that constrains `?g` to be `g`:

```
(defun-sk2 def-?g (?g) ()
  (forall (y1 y2) (equal (?g y1 y2) (g y1 y2)))
  :rewrite :direct)
(defun2 spec4[?h_?f_?g] (?h ?f ?g) ()
  (and (def-?h-fold[?f_?g]) (def-?f) (def-?g)))
(defthm step4-lemma (implies (def-?g) (consp-io[?g]))
  :hints (("Goal" :in-theory '(consp-io[?g] consp-io[g]-necc
                               consp-io[g]! def-?g-necc))))
(defthm step4 (implies (spec4[?h_?f_?g]) (spec3[?h_?f_?g]))
  :hints (("Goal" :in-theory '(spec3[?h_?f_?g] spec4[?h_?f_?g] step4-lemma))))
```

Step 5 Substituting the solutions `f` and g into `fold[?f_?g]` yields a solution for `?h`:

```
(defun-inst h (fold[?f_?g] (?f . f) (?g . g)))
(defun-sk2 def-?h (?h) () (forall x (equal (?h x) (h x))) :rewrite :direct)
```

The conjunct `def-?h-fold[?f_?g]` of `spec4[?h_?f_?g]` is replaced with `def-?h`, which is equivalent to `def-?h-fold[?f_?g]` given the conjuncts `def-?f` and `def-?g`:

```
(defun2 spec5[?h_?f_?g] (?h ?f ?g) () (and (def-?h) (def-?f) (def-?g)))
(defthm step5-lemma
  (implies (and (def-?f) (def-?g)) (equal (h x) (fold[?f_?g] x)))
  :hints (("Goal" :in-theory '(h fold[?f_?g] def-?f-necc def-?g-necc))))
(defthm step5 (implies (spec5[?h_?f_?g]) (spec4[?h_?f_?g]))
  :hints (("Goal" :in-theory '(spec4[?h_?f_?g] spec5[?h_?f_?g]
                               def-?h-fold[?f_?g] def-?h-necc step5-lemma))))
```

This concludes the derivation: `spec[?h_?f_?g]` provides executable solutions for `?h`, `?f`, and `?g`. The resulting implementation is $\langle \mathtt{h}, \mathtt{f}, \mathtt{g} \rangle$. Chaining the implications of the step correctness theorems shows that these solutions satisfy the requirements specification:

```
(defthm chain[?h_?f_?g] (implies (spec5[?h_?f_?g]) (spec[?h]))
  :hints (("Goal" :in-theory '(step1 step2 step3 step4 step5))))
```

More explicitly, instantiating the end-to-end implication shows that `h` satisfies the requirements specification:

```
(defun-inst def-h (def-?h (?h . h)))
(defun-inst def-f (def-?f (?f . f)))
(defun-inst def-g (def-?g (?g . g)))
(defun-inst spec5[h_f_g] (spec5[?h_?f_?g] (?h . h) (?f . f) (?g . g)))
(defun-inst spec[h] (spec[?h] (?h . h)))
(defthm-inst chain[h_f_g] (chain[?h_?f_?g] (?h . h) (?f . f) (?g . g)))
(defthm spec5[h_f_g]! (spec5[h_f_g])
  :hints (("Goal" :in-theory '(spec5[h_f_g]))))
(defthm spec[h]! (spec[h])
  :hints (("Goal" :in-theory '(chain[h_f_g] spec5[h_f_g]!))))
```

3 Related Work

The `instance-of-defspec` tool [14] and the `make-generic-theory` tool [17] automatically generate instances of functions and theorems that reference functions constrained via encapsulation [15], by replacing the constrained functions with functions that satisfy the constraints. The instantiation mechanisms of these tools are similar to the ones of SOFT; constrained functions in these tools parallel function variables in SOFT. However, in SOFT function variables are unconstrained; constraints on them are expressed via second-order predicates (typically with quantifiers), and the same function variables can be used as parameters of different constraining predicates. Unlike SOFT, `instance-of-defspec` and `make-generic-theory` do not handle choice and quantifier functions, and do not generate termination proofs for recursive function instances. SOFT generates one function or theorem instance at a time, while `instance-of-defspec` and `make-generic-theory` can generate many. These two tools are more suited to developing and instantiating abstract and parameterized theories; SOFT is more suited to mimic second-order logic notation.

The `:consider` hint [19] heuristically generates functional instantiations to help prove given theorems. SOFT generates function and theorem instances for given replacements of function variables; from these replacements, the necessary functional instantiations are generated automatically.

The `def-functional-instance` tool in the ACL2 community books generates theorem instances for given replacements of functions. This tool has more general use than SOFT's `defthm-inst`, but it

requires a complete functional instantiation, while `defthm-inst` only requires replacements for function variables.

Wrapping existing events to record information for later use (as done by SOFT's `defunvar`, `defun2`, `defchoose2`, and `defun-sk2`) has precedents. For example, the `def:un-sk` tool [11] is a wrapper of `defun-sk` that records information to help prove theorems involving quantifiers. It may be useful to combine `def:un-sk` with SOFT's `defun-sk2` wrapper.

There are several tools to generate functions and theorems according to certain patterns, such as `std::deflist` in the ACL2 standard library and `fty::deflist` in the FTY library [23]. These tools may use SOFT to generate some of the functions and theorems as instances of pre-defined second-order functions and theorems.

A general-purpose theorem prover like ACL2 can represent a variety of specification and refinement formalisms, e.g. [1, 2, 12, 13, 18, 20, 22]; derivations can be carried out within the logic. But given the close ties to Applicative Common Lisp, a natural approach to program refinement in ACL2 is to specify requirements on one or more target ACL2 functions, and progressively strengthen the requirements until the functions are executable and performant.

Alternatives to SOFT's second-order predicates, for specifying requirements on ACL2 functions, include `encapsulate` (possibly via the wrappers `defspec` and `defabstraction` in the ACL2 community books), `defaxiom`, and `defchoose`. But these are not as suited to program refinement:

- An `encapsulate` involves exhibiting witnesses to the consistency of the requirements, which amounts to writing an implementation and proving it correct. But it is the purpose of program refinement to construct an implementation and its correctness proof.
- A `defaxiom` obviates witnesses but may introduce logical inconsistency.
- A `defchoose` obviates witnesses and is logically conservative, but:
  - It expresses requirements on single functions, necessitating the combination of multiple target functions into one.
  - It expresses requirements on function results (the bound variables) with respect to function arguments (the free variables), but not requirements involving different results and different arguments, such as injectivity, non-interference [10], and other hyperproperties [7].
  - It prescribes underspecified but fixed function results. For example, there is no clear refinement relation between the function introduced as `(defchoose f (y) (x) (> y x))` and the function introduced as `(defun g (x) (+ x 1))`.

In contrast, a second-order predicate can specify any kind of requirements, on multiple functions, maintaining logical consistency, and doing so without premature witnesses.

In the derivation in Section 2.2, the use and instantiation of the generic folding function on binary trees is an example of the application of algorithm schemas in program refinement, as in [21] but here realized via second-order functions and theorems. Second-order functions express algorithm schemas, and second-order theorems show their correctness under suitable conditions on the function parameters. Applying a schema adds a constraint that defines a target function to use the schema, and introduces simpler target functions corresponding to the function parameters, constrained to satisfy the conditions for the correctness of the schema.

A refinement step from a specification $spec_j$ can be performed manually, by writing down $spec_{j+1}$ and proving `(implies (`$spec_{j+1}$`) (`$spec_j$`))`. It is sometimes possible to generate $spec_{j+1}$ from $spec_j$, along with a proof of `(implies (`$spec_{j+1}$`) (`$spec_j$`))`, using automated transformation techniques. Automated transformations may require parameters to be provided and applicability conditions to be proved,

but should generally save effort and make derivations more robust against changes in requirements specifications. At Kestrel Institute, we are developing ACL2 libraries of automated transformations for program refinement.

4 Future Work

Guards `defun-inst` could be extended with the option to override the default guard $\Sigma(guard)$ with a different *guard'*, generating the proof obligation `(implies` *guard'* $\Sigma(guard)$`)`. This would be useful in at least two situations.

A first situation is when the function instance has more guard conditions to verify than the second-order function being instantiated, due to the replacement of a function parameter (which has no guards) with a function that has guards. Providing a stronger guard to the function instance would enable the verification of the additional guard conditions. For example, an instance `quad[cdr]` of `quad[?f]` from Section 1.2.1 could be supplied with the guard `(true-listp x)`.

A second situation is when the guard of the second-order function being instantiated includes conditions on function parameters that involve a quantifier, e.g. the condition that the binary operation `?op` of a generic folding function over lists is closed over the type `?p` of the list elements. Instantiating `?p` with `natp` and `?op` with `binary-+` satisfies the condition, but $\Sigma(guard)$ still includes a quantifier that makes the instance of the folding function non-executable. Supplying a *guard'* that rephrases $\Sigma(guard)$ to omit the satisfied closure condition would solve the problem. As guard obligations on individual parameters are relieved when functions are applied to terms in a term, it makes sense to relieve guard obligations on function parameters when second-order functions are "applied" to functions in `defun-inst`.

`defun-inst` could also be extended with the ability to use the instances of the verified guard conditions of the second-order function being instantiated, to help verify the guard conditions of the function instance. This may completely verify the guards of the instance, when no guard overriding is needed.

Partial Functions SOFT could be extended with a macro `defpun2` to introduce partial second-order functions, mimicked by partial first-order functions introduced via `defpun` [16]. `defun-inst` could be extended to generate not only partial function instances, but also total function instances when the instantiated `:domain` or `:gdomain` restrictions are theorems. Partial second-order functions would be useful, in particular, to define recursive algorithm schemas whose measures and whose argument updates in recursive calls are, or depend on, function parameters. An example is a general divide-and-conquer schema.[4]

Mutual Recursion SOFT could be extended with a macro `mutual-recursion2` to introduce mutually recursive plain second-order functions (with `defun2`), mimicked by mutually recursive first-order functions introduced via `mutual-recursion`. `defun-inst` could be extended to generate instances of mutually recursive second-order functions.

Lambda Expressions `defun-inst` and `defthm-inst` could be extended to accept instantiations that map function variables to lambda expressions, similarly to `:functional-instance`.

[4] The folding function from Section 1.2.1 is a divide-and-conquer schema specialized to binary trees.

Instantiation Transitivity If *sof'* is introduced as the Σ instance of *sof*, and *f* is introduced as the Σ' instance of *sof'*, then *f* should be the Σ" instance of *sof*, where Σ" is a suitably defined composition of Σ and Σ'. Currently `defun-inst` does not record *f* as an instance of *sof* when *f* is introduced, but it could be extended to do so. With this extension, `injective[quad[wrap]]` in Section 1.5 would be the `((?f . quad[wrap]))` instance of `injective[?f]` in Section 1.2.3.

In a related but different situation, given *sof*, *sof'*, *f*, Σ, Σ', and Σ" as above, but with *f* introduced as the Σ" instance of *sof*, and *sof'* introduced as the Σ instance of *sof*, in either order (i.e. *f* then *sof'*, or *sof'* then *f*), then *f* should be the Σ' instance of *sof'*. Currently `defun-inst` does not record *f* as an instance of *sof'* when *f* (after *sof'*) or *sof'* (after *f*) is introduced, but could be extended to do so. With this extension, if `injective[quad[wrap]]` were introduced as the `((?f . quad[wrap]))` instance of `injective[?f]`, and `injective[quad[?f]]` were introduced as the `((?f . quad[?f]))` instance of `injective[?f]` as in Section 1.3, then `injective[quad[wrap]]` would be the `((?f . wrap))` instance of `injective[quad[?f]]`.

An alternative to these two extensions of `defun-inst` is to extend SOFT with a macro to claim that an existing instance of a second-order function is also an instance of another second-order function according to a given instantiation. The macro would check the claim (by applying the instantiation and comparing the result with the function) and extend the table of instances of second-order functions if the check succeeds. In the first scenario above, the macro would be used to claim that *f* is the Σ" instance of *sof*; in the second scenario above, the macro would be used to claim that *f* is the Σ' instance of *sof'*.

Function Variable Constraints Currently the only constraints on function variables are their types. `defunvar` could be extended to accept richer signatures for function variables, with multiple-value results and single-threaded arguments and results. `defun-inst` and `defthm-inst` would then be extended to check that instantiations satisfy these additional constraints. A more radical extension would be to attach logical constraints to certain function variables, as in encapsulations.

Automatic Instances As explained in Section 1.3, when an instantiation is applied to a term, the table of instances of second-order functions is consulted to find replacements for certain second-order functions, and the application of the instantiation fails if replacements are not found. Thus, all the needed instances must be introduced before applying the instantiation, e.g. in Section 1.5 the two `defun-insts` had to be supplied before the last `defthm-inst`. SOFT could be extended to generate automatically the needed instances of second-order functions.

SOFT could also be extended with a macro `defthm2` to prove a second-order theorem via `defthm` and to record the theorem in a table, along with information about the involved second-order functions. `defun-inst` could be extended with the option to generate instances of the second-order theorems that involve the second-order function being instantiated. `defthm2` could include the option to generate instances of the theorem that correspond to the known instances of the second-order functions that the theorem involves. These extensions would reduce the use of explicit `defthm-insts`.

The convention of including function variables in square brackets in the names of second-order functions and theorems, could be exploited to name the automatically generated function and theorem instances, as suggested by the examples throughout the paper.

Other Events SOFT could be extended to provide second-order counterparts of other function and theorem introduction events, e.g. `define`, `defines`, and `defrule` in the ACL2 community books.

References

[1] Martín Abadi & Leslie Lamport (1991): *The Existence of Refinement Mappings*. *Journal of Theoretical Computer Science* 82(2), doi:10.1016/0304-3975(91)90224-P.

[2] Jean-Raymond Abrial (1996): *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, doi:10.1017/CBO9780511624162.

[3] *The ACL2 Theorem Prover*. `http://www.cs.utexas.edu/~moore/acl2`.

[4] Peter B. Andrews (2002): *An Introduction to Mathematical Logic and Type Theory: To Thruth Through Proof*. Springer, doi:10.1007/978-94-015-9934-4.

[5] R. S. Boyer, D. M. Goldschlag, M. Kaufmann & J S. Moore (1991): *Functional Instantiation in First Order Logic*. Technical Report 44, Computational Logic Inc.

[6] Alonzo Church (1940): *A Formulation of the Simple Theory of Types*. *The Journal of Symbolic Logic* 5(2), doi:10.2307/2266170.

[7] Michael Clarkson & Fred Schneider (2010): *Hyperproperties*. *Journal of Computer Security* 18(6), doi:10.3233/JCS-2009-0393.

[8] Alessandro Coglio (2014): *Pop-Refinement*. *Archive of Formal Proofs*. `http://afp.sf.net/entries/Pop_Refinement.shtml`, Formal proof development.

[9] Edsger W. Dijkstra (1968): *A Constructive Approach to the Problem of Program Correctness*. *BIT* 8(3), doi:10.1007/BF01933419.

[10] Joseph Goguen & José Meseguer (1982): *Security Policies and Security Models*. In: *Proc. IEEE Symposium on Security and Privacy*, doi:10.1109/SP.1982.10014.

[11] David Greve (2009): *Automated Reasoning with Quantified Formulae*. In: *Proc. 8th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2009)*, doi:10.1145/1637837.1637855.

[12] C. A. R. Hoare (1972): *Proof of Correctness of Data Representations*. *Acta Informatica* 1(4), doi:10.1007/BF00289507.

[13] Cliff Jones (1990): *Systematic Software Development using VDM*, second edition. Prentice Hall.

[14] Sebastiaan J. C. Joosten, Bernard van Gastel & Julien Schmaltz (2013): *A Macro for Reusing Abstract Functions and Theorems*. In: *Proc. 11th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2013)*, doi:10.4204/EPTCS.114.3.

[15] Matt Kaufmann & J Strother Moore (2001): *Structured Theory Development for a Mechanized Logic*, doi:10.1023/A:1026517200045.

[16] Panagiotis Manolios & J Strother Moore (2003): *Partial Functions in ACL2*, doi:10.1023/B:JARS.0000009505.07087.34.

[17] F. J. Martìn-Mateos, J. A. Alonso, M. J. Hidalgo & J. L. Ruiz-Reina (2002): *A Generic Instantiation Tool and a Case Study: A Generic Multiset Theory*. In: *Proc. 3rd International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2002)*.

[18] Robin Milner (1971): *An Algebraic Definition of Simulation between Programs*. Technical Report CS-205, Stanford University.

[19] J Strother Moore (2009): *Automatically Computing Functional Instantiations*. In: *Proc. 8th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2009)*, doi:10.1145/1637837.1637839.

[20] Carroll Morgan (1998): *Programming from Specifications*, second edition. Prentice Hall.

[21] Douglas R. Smith (1999): *Mechanizing the Development of Software*. In Manfred Broy, editor: *Calculational System Design, Proc. Marktoberdorf Summer School*, IOS Press.

[22] J. M. Spivey (1992): *The Z Notation: A Reference Manual*, second edition. Prentice Hall.

[23] Sol Swords & Jared Davis (2015): *Fix Your Types*. In: *Proc. 13th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2015)*.

Fourier Series Formalization in ACL2(r)

Cuong K. Chau
Department of Computer Science
The University of Texas at Austin
Austin, TX, USA
ckcuong@cs.utexas.edu

Matt Kaufmann
Department of Computer Science
The University of Texas at Austin
Austin, TX, USA
kaufmann@cs.utexas.edu

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

We formalize some basic properties of Fourier series in the logic of ACL2(r), which is a variant of ACL2 that supports reasoning about the real and complex numbers by way of non-standard analysis. More specifically, we extend a framework for formally evaluating definite integrals of real-valued, continuous functions using the Second Fundamental Theorem of Calculus. Our extended framework is also applied to functions containing free arguments. Using this framework, we are able to prove the orthogonality relationships between trigonometric functions, which are the essential properties in Fourier series analysis. The sum rule for definite integrals of indexed sums is also formalized by applying the extended framework along with the First Fundamental Theorem of Calculus and the sum rule for differentiation. The Fourier coefficient formulas of periodic functions are then formalized from the orthogonality relations and the sum rule for integration. Consequently, the uniqueness of Fourier sums is a straightforward corollary.

We also present our formalization of the sum rule for definite integrals of infinite series in ACL2(r). Part of this task is to prove the Dini Uniform Convergence Theorem and the continuity of a limit function under certain conditions. A key technique in our proofs of these theorems is to apply the *overspill principle* from non-standard analysis.

1 Introduction

In this paper, we present our efforts in formalizing some basic properties of Fourier series in the logic of ACL2(r), which is a variant of ACL2 that supports reasoning about the real and complex numbers via non-standard analysis [8, 12]. In particular, we describe our formalization of the Fourier coefficient formulas for periodic functions and the sum rule for definite integrals of infinite series. The formalization of Fourier series will enable interactive theorem provers to reason about systems modeled by Fourier series, with applications to a wide variety of problems in mathematics, physics, electrical engineering, signal processing, and image processing.

We do not claim to be developing new mathematics. However, as far as we know the mechanized formalizations and proofs presented in this paper are new. The research contributions of this paper are twofold: a demonstration that a mechanized proof assistant, in particular ACL2(r), can be used to verify properties of Fourier series; and infrastructure to support that activity, which we expect to be reusable for future ACL2(r) verifications of continuous mathematics. Our formalizations presented in the paper assume that there exists a Fourier series, i.e., a (possibly infinite) sum of sines and cosines for any periodic function. Future work could include proving convergence of the Fourier series for any suitable periodic function.

The proofs of *Fourier coefficient formulas* depend on the *orthogonality relationships between trigonometric functions* and the *sum rule for integration of indexed sums*. A key tool for proving these properties is the Second Fundamental Theorem of Calculus (FTC-2). Cowles and Gamboa [5] implemented a framework for formally evaluating definite integrals of real-valued continuous functions using FTC-2.

M. Kaufmann and D. Rager (Eds.): ACL2 Workshop 2015 (ACL2 2015).
EPTCS 192, 2015, pp. 35–51, doi:10.4204/EPTCS.192.4

However, their framework is restricted to unary functions, while formalizing Fourier coefficient formulas requires integration for indexed families of functions $f_n(x)$, which we represent as $f(x,n)$. We call such n a *free argument*. Hence, we extend the FTC-2 framework of Cowles and Gamboa to apply to functions with free arguments. We call the extended framework the *FTC-2 evaluation procedure*. One may expect the usual ACL2 *functional instantiation* mechanism to apply, by using pseudo-lambda expressions [1] to handle the free arguments. However, in ACL2(r) there are some technical issues and restrictions on the presence of free arguments in functional substitutions, which make functional instantiation not trivial [4]. We describe these issues in detail and show how we deal with them in Section 4. Once the FTC-2 evaluation procedure is built, we can use it to prove the orthogonality relationships between trigonometric functions. The sum rule for definite integrals of indexed sums is also formalized by applying the FTC-2 evaluation procedure along with the First Fundamental Theorem of Calculus (FTC-1) and the sum rule for differentiation. The Fourier coefficient formulas for periodic functions are then verified using the orthogonality relations and the sum rule for integration. Consequently, the *uniqueness of Fourier sums* is a straightforward corollary of the Fourier coefficient formulas.

The other main contribution of our work is the formalization of the *sum rule for definite integrals of infinite series* under two different conditions. This problem deals with the *convergence* notion of a sequence of functions. We consider two types of convergence: *pointwise convergence* and *uniform convergence*. Our formalization requires that a sequence of partial sums of real-valued continuous functions converges uniformly to a *continuous limit function* on the interval of interest. We approach this requirement in two ways, corresponding to two different conditions. One way is to prove that if a sequence of continuous functions converges pointwise on a closed and bounded interval, then it converges uniformly on that interval, given that the sequence is monotonic and the limit function is continuous. This is known as the *Dini Uniform Convergence Theorem* [15]. Another way is to prove that if a sequence of continuous functions is not required to be monotonic but converges uniformly to some limit function on the interval of interest, then the limit function is also continuous on that interval. A key technique in our proofs for both cases is to apply the *overspill principle* from non-standard analysis [9, 13]. Thus, we also formalize the overspill principle in ACL2(r) and apply this principle to prove Dini's theorem and the continuity of the limit function as mentioned.

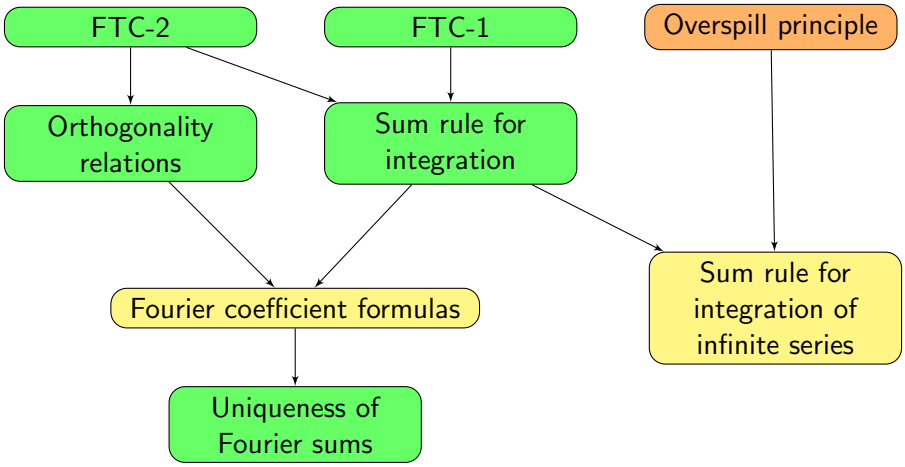


Figure 1. Overview of the formalization of the Fourier coefficient formulas and the sum rule for definite integrals of infinite series.

Figure 1 gives an overview of the work presented in this paper. The remainder of the paper is organized as follows. Section 2 reviews some basic notions of non-standard analysis in ACL2(r) that we

use later in the paper. Section 3 reviews two versions of the Fundamental Theorem of Calculus that we extend to support our Fourier series formalization. Section 4 describes the FTC-2 evaluation procedure as an extended framework for applying FTC-2 to functions with free arguments. The formalization of the orthogonality relations for trigonometric functions and the sum rule for definite integrals of indexed sums are described in Sections 5 and 6 respectively. The formalization of the Fourier coefficient formulas and the uniqueness of Fourier sums are described in Section 7. The preceding results apply to finite series, but as we look ahead to dealing with infinite Fourier series, we take a step in Section 8, which presents our formalization of the sum rule for definite integrals of infinite series. Finally, Section 9 concludes the paper and points out some possible future work.

2 Basic Non-Standard Analysis Notions in ACL2(r)

Here we review basic notions of non-standard analysis in ACL2(r) that are used in the remainder of this paper. All notions introduced here are considered *non-classical*, while functions whose definitions do not depend on any of these notions are *classical*. Let x be a real number.

- A primitive notion is that x is *standard*, which intuitively means that x is a "traditional" real number. In particular, x is standard if it can be defined. For example, 1, -2, 3.65, π, e^5, and $\sqrt{2}$ are standard. A natural number is considered standard if it is finite, otherwise it is non-standard. We will refer to the standard notion of natural numbers when stating the overspill principle in Section 8. We feel free to *relativize* our quantifiers. For example, "$\forall^{st} n \ldots$" means "for all standard $n \ldots$", and "$\exists^{\neg st} n \ldots$" means "there exists non-standard $n \ldots$".
- x is `i-small` (*infinitesimal*) iff $|x| < r$ for all positive standard reals r.
- x is `i-large` iff $|x| > r$ for all positive standard reals r.
- x is `i-limited` (*finite*) iff $|x| < r$ for some positive standard real r.
- x is `i-close` ($\approx$) to a real y iff $(x - y)$ is `i-small`.
- Suppose x is `i-limited`. Then `standard-part`(x), or simply st(x), is the *unique standard real* that is `i-close` to x.

3 Fundamental Theorem of Calculus

This section reviews two versions of the Fundamental Theorem of Calculus that we need to extend to functions with free arguments, as part of our Fourier series formalization. The two versions are sometimes called the First and Second Fundamental Theorem of Calculus.

First Fundamental Theorem of Calculus (FTC-1): Let f be a real-valued continuous function on the interval $[a,b]$. We can then define a corresponding function $g(x)$ as follows: $g(x) = \int_a^x f(t)dt$. Then $g'(x) = f(x)$ for all $x \in [a,b]$.

Second Fundamental Theorem of Calculus (FTC-2): If f is a real-valued continuous function on $[a,b]$ and g is an antiderivative of f on $[a,b]$, i.e., $g'(x) = f(x)$ for all $x \in [a,b]$, then

$$\int_a^b f(x)dx = g(b) - g(a).$$

In the next two sections, we extend FTC-2 to functions with free arguments and apply it to prove the orthogonality relations of trigonometric functions, respectively. The extension of FTC-1 and its application to the sum rule for definite integrals of indexed sums is described in Section 6.

4 FTC-2 Evaluation Procedure

This section describes how we apply the FTC-2 theorem to evaluate definite integrals of real-valued continuous functions f in terms of their antiderivatives g, even when f and g contain *free arguments*, that is, arguments other than the variable with respect to which we perform integration or differentiation. In particular, we extend the existing FTC-2 framework [5] to functions with free arguments, and call the extended framework the *FTC-2 evaluation procedure*. This procedure consists of the following steps:

- Prove that f returns real values on $[a,b]$.
- Prove that f is continuous on $[a,b]$.
- Specify a real-valued antiderivative g of f and prove that f is the derivative of g on $[a,b]$; i.e., prove that g returns real values and $g'(x) = f(x)$ for all $x \in [a,b]$.
- Formalize the integral of f on $[a,b]$ as the Riemann integral.
- Evaluate the integral of f on $[a,b]$ in terms of g by applying the FTC-2 theorem.

The first two steps are trivial in comparison to the last three. In the following subsections, we describe the challenges manifest in the last three steps and how we tackle them.

4.1 Automatic Differentiator

In order to apply the FTC-2 evaluation procedure to evaluate the definite integral of a function f, we need to specify and prove the correctness of a real-valued antiderivative g of f. The specifying task can be done by appealing to a computer algebra system such as *Mathematica* [16]. Notably, we must mechanically check in ACL2(r) that f is indeed the derivative of g. Fortunately, we don't have to prove this manually for every function. An *automatic differentiator* (AD) implemented by Reid and Gamboa [6, 7] symbolically computes the derivative f of the input function g and automatically derives a proof demonstrating the correctness of the differentiation, i.e., automatically proves the following formula:

$$f(x) \approx \frac{g(x) - g(y)}{x - y},$$

for all x and y in the domain of g such that x is standard, $x \approx y$, but $x \neq y$. For example, the user can employ the AD to prove that $f(x) = n\cos(nx)$ is the derivative of $g(x) = \sin(nx)$ with respect to x, by calling the macro `defderivative` with the input function g as follows:

```
(defderivative sine-derivative
  (acl2-sine (* n x)))
```

The following theorem is then introduced and proved automatically:

```
(defthm sine-derivative
  (implies (and (acl2-numberp x)
                (acl2-numberp (* n x))
                (acl2-numberp y)
                (acl2-numberp (* n y))
                (standardp x)
                (standardp n) (acl2-numberp n)
                (i-close x y) (not (equal x y)))
```

```
(i-close (/ (- (acl2-sine (* n x))
               (acl2-sine (* n y)))
            (- x y))
         (* (acl2-cosine (* n x))
            (+ (* n 1) (* x 0))))))
```

The AD requires using the symbol x as the name of the variable with respect to which the (partial) derivative is computed. Notice that the hypotheses `(acl2-numberp (* n x))` and `(acl2-numberp (* n y))` in the above theorem are redundant since they can be implied from the set of hypotheses `(acl2-numberp x)`, `(acl2-numberp y)` and `(acl2-numberp n)`. In addition, the above theorem states that the derivative of $\sin(nx)$ is $\cos(nx)(n*1+x*0)$, which indeed equals $n\cos(nx)$. This AD does not perform such simplifications. Nevertheless, the user can easily prove the desired theorem from the one generated by the macro `defderivative`.

4.2 Formalizing the Riemann Integral with Free Arguments

We formalize the definite integral of a function as the Riemann integral, following the same method as implemented by Kaufmann [11]. When functions contain free arguments, this formalization encounters a problem with functional instantiations of non-classical theorems containing these functions. We will describe the problem in detail and how we deal with it. Let's consider the following definition of the Riemann integral of a *unary* function, which uses an ACL2(r) utility, `defun-std` [4], for introducing classical functions defined in terms of non-classical functions. Note that `defun-std` defines a function which is only guaranteed to satisfy its definition on standard inputs.

```
(defun-std strict-int-f (a b)
  (if (and (inside-interval-p a (f-domain))
           (inside-interval-p b (f-domain))
           (< a b))
      (standard-part (riemann-f (make-small-partition a b)))
    0))
```

The form above introduces the Riemann integral of a function f as a classical function, even though it contains two non-classical functions, `standard-part` and `make-small-partition`[1]. The proof obligation here is to prove the integral returns standard values with standard inputs. More specifically, we need to prove that the standard part of the Riemann sum of f, for any partition of $[a,b]$ with standard endpoints into infinitesimal-length subintervals, returns standard values. This is true only if that Riemann sum is limited. In fact, for a generic real-valued continuous *unary* function `rcfn`, this limited property was proven for a corresponding Riemann sum, as follows [11].

```
(defthm limited-riemann-rcfn-small-partition
  (implies (and (standardp a)
                (standardp b)
                (inside-interval-p a (rcfn-domain))
                (inside-interval-p b (rcfn-domain))
                (< a b))
           (i-limited (riemann-rcfn (make-small-partition a b)))))
```

[1] We use the non-classical function `make-small-partition` to partition a closed and bounded interval into subintervals each of infinitesimal length.

We are now interested in extending the above theorem for functions containing free arguments using functional instantiation with pseudo-lambda expressions. Unfortunately, free arguments are not allowed to occur in pseudo-lambda expressions in the functional substitution since the theorem we are trying to instantiate is non-classical and the functions we are trying to instantiate are classical; the following example shows why this requirement is necessary [4]. For an arbitrary classical function $f(x)$, the following is a theorem.

$$standardp(x) \Rightarrow standardp(f(x))$$

Substitution of $\lambda(x).(x+y)$ for f into the above formula yields the formula

$$standardp(x) \Rightarrow standardp(x+y)$$

which is not valid, since the free argument y can be non-standard.

Instead of using functional instantiation, we prove the limited property of Riemann sums (as discussed above) from scratch by applying the following theorem.

Theorem 1 (The boundedness of Riemann sums [11])**.** *Assume that there exist finite values m and M such that*

$$m \leq f(t) \leq M, \text{ for all } t \in [a,b].$$

Then the Riemann sum of f over $[a,b]$ with any partition $P = \{x_0, x_1, \ldots, x_n\}$ is bounded by

$$m(b-a) \leq \sum_{i=1}^{n} f(t_i)(x_i - x_{i-1}) \leq M(b-a)$$

where $t_i \in [x_{i-1}, x_i]$, $x_0 = a$, and $x_n = b$.

From Theorem 1, proving the Riemann sum of f over $[a,b]$ is bounded reduces to proving f is bounded on that interval. Given a *specific* real-valued continuous function f, it is usually straightforward to specify the bounds of f on a closed and bounded interval. The problem becomes more challenging when applying to *generic* real-valued continuous functions since it is impossible to find either their minimum or maximum. However, the boundedness of these functions on a closed and bounded interval still holds by the *extreme value theorem*. But again, this was just proven for *unary* functions [5]. We also want to apply this property to functions with free arguments. Our solution at this point is to re-prove the extreme value theorem and consequently the limited property of Riemann sums for generic functions with free arguments. Since the number of free arguments is varied, it would be troublesome to prove the same properties independently for each number of free arguments. Indeed, we just need to add only one *extra* argument representing a list of the free arguments to the constrained functions and re-prove the concerned non-classical theorems. The necessary hypotheses for the extra argument can be added throughout the proof development. Note that non-classical theorems proven for the new constrained functions with only one extra argument added can also be derived for functions with an arbitrary number of free arguments, using functional and ordinary instantiation. (See lemmas `limited-riemann-f-small-partition-lemma` and `limited-riemann-f-small-partition` below for an example of how this works.) The question is how can we avoid the problem of the appearance of free arguments in functional instantiations of non-classical theorems as described above? The trick is to treat the extra argument in the constrained functions as a list of free arguments. Thus, no free argument appears in the functional instantiations. To illustrate the proposed technique, let us investigate the constrained function `rcfn-2` below. It contains one main argument `x` and one extra argument `arg`.

```
(encapsulate
 ((rcfn-2 (x arg) t)
  (rcfn-2-domain () t))

 ;; Our witness real-valued continuous function is the
 ;; identity function of x. We ignore the extra argument arg.

 (local (defun rcfn-2 (x arg) (declare (ignore arg)) (realfix x)))
 (local (defun rcfn-2-domain () (interval nil nil)))

 ... ;; Non-local theorems about rcfn-2 and rcfn-2-domain
 )
```

We then prove the extreme value theorem for `rcfn-2` and consequently the limited property of the Riemann sum of `rcfn-2`, using the same proofs for the case of *unary* function `rcfn` existing in the ACL2 community books [2], file `books/nonstd/integrals/continuous-function.lisp`. The limited property of the Riemann sum of `rcfn-2` is stated as follows:

```
(defthm limited-riemann-rcfn-2-small-partition
  (implies (and (standardp arg)
                (standardp a)
                (standardp b)
                (inside-interval-p a (rcfn-2-domain))
                (inside-interval-p b (rcfn-2-domain))
                (< a b))
           (i-limited (riemann-rcfn-2 (make-small-partition a b) arg))))
```

As claimed, the above non-classical theorem can also be applied to functions with an arbitrary number of free arguments, using the trick we describe in the following example. In this example, the function $f(x,m,n)$ contains two free arguments m and n. Then, the parameter `arg` in the above theorem should be considered as the list `(list m n)`. Having said that, we first need to prove a lemma stating that *every element in a standard list is standard*. This can be proven easily by using `defthm-std` [4].

```
(defthm-std standardp-nth-i-arg
  (implies (and (standardp arg)
                (standardp i))
           (standardp (nth i arg)))
  :rule-classes (:rewrite :type-prescription))
```

The functional instantiation with pseudo-lambda expressions can now be applied to prove the limited property of the Riemann sum of f as follows.

```
(1)  (defthm limited-riemann-f-small-partition-lemma
       (implies (and (standardp arg)
                     (standardp a)
                     (standardp b)
                     (inside-interval-p a (f-domain))
                     (inside-interval-p b (f-domain))
                     (< a b))
                (i-limited (riemann-f (make-small-partition a b)
```

```
                                    (nth 0 arg)
                                    (nth 1 arg))))
      :hints (("Goal"
               :by (:functional-instance
                    limited-riemann-rcfn-2-small-partition
                    (rcfn-2 (lambda (x arg)
                              (f x (nth 0 arg) (nth 1 arg))))
                    (rcfn-2-domain f-domain)
                    (map-rcfn-2
                     (lambda (p arg)
                       (map-f p (nth 0 arg) (nth 1 arg))))
                    (riemann-rcfn-2
                     (lambda (p arg)
                       (riemann-f p (nth 0 arg) (nth 1 arg)))))))))
```

Note that the functional instantiation in the above lemma does not contain any free arguments. However, this lemma constrains the two free arguments to be members of a list. In order to eliminate this constraint, we need a lemma stating that *a list of length two is standard if both of its elements are standard*. Again, we can prove this using `defthm-std`.

```
(defthm-std standardp-list
  (implies (and (standardp m)
                (standardp n))
           (standardp (list m n)))
  :rule-classes (:rewrite :type-prescription))
```

We are finally able to prove the desired theorem as an instance of the lemma (1).

```
(defthm limited-riemann-f-small-partition
  (implies (and (standardp m)
                (standardp n)
                (standardp a)
                (standardp b)
                (inside-interval-p a (f-domain))
                (inside-interval-p b (f-domain))
                (< a b))
           (i-limited (riemann-f (make-small-partition a b) m n)))
  :hints (("Goal"
             :use (:instance limited-riemann-f-small-partition-lemma
                             (arg (list m n))))))
```

4.3 Applying FTC-2 to Functions with Free Arguments

The FTC-2 theorem was stated and proven in the ACL2 community books for generic *unary* functions as follows [5]:

```
(defthm ftc-2
  (implies (and (inside-interval-p a (rcdfn-domain))
                (inside-interval-p b (rcdfn-domain)))
```

```
(equal (int-rcdfn-prime a b)
       (- (rcdfn b) (rcdfn a)))))
```

Again, we would like to apply this theorem for functions with free arguments via functional instantiation. Since this theorem is classical, free arguments are allowed to occur in pseudo-lambda expressions of a functional substitution as long as classicalness is preserved [4]. Through functional instantiation with pseudo-lambda terms, we encounter several proof obligations that require free arguments to be standard. Unfortunately, attempting to add this assumption to pseudo-lambda terms, e.g., `(lambda (x) (if (standardp n) (f x n) (f x 0)))`, is not allowed in ACL2(r) since the terms become non-classical by using the non-classical function `standardp`, violating the classicalness requirement. To deal with this issue of functional instantiation, we propose a technique using an encapsulate event with *zero-arity classical functions* (constants) representing free arguments. Since the zero-arity functions are classical, they must return standard values. Using this technique, we can instantiate the FTC-2 theorem to evaluate the definite integral of a function containing free arguments in terms of its antiderivative. For example, suppose we want to apply the FTC-2 theorem to a real-valued continuous function $f(x,n)$, where n is a free argument of type integer. Also suppose that g is an antiderivative of f. Our proposed technique consists of four steps as described below:

- Step 1: Define an encapsulate event that introduces zero-arity classical function(s) representing free argument(s).

```
(encapsulate
 (((n) => *))
 (local (defun n () 0))
 (defthm integerp-n
   (integerp (n))
   :rule-classes :type-prescription))
```

- Step 2: Prove that the zero-arity classical function(s) return standard values using `defthm-std`.

```
(defthm-std standardp-n
  (standardp (n))
  :rule-classes (:rewrite :type-prescription))
```

- Step 3: Prove the main theorem, modified by replacing the free argument(s) with the corresponding zero-arity function(s) introduced in step 1. Without free argument(s), the functional instantiation can be applied straightforwardly.

```
(defthm f-ftc-2-lemma
  (implies (and (inside-interval-p a (g-domain))
                (inside-interval-p b (g-domain)))
           (equal (int-f a b (n))
                  (- (g b (n))
                     (g a (n)))))
  :hints (("Goal"
           :by (:functional-instance
                 ftc-2
                 (rcdfn
                  (lambda (x) (g x (n))))
                 (rcdfn-prime
                  (lambda (x) (f x (n)))))
```

```
(rcdfn-domain g-domain)
... ;; Instantiate other constrained
    ;; functions similarly.
(int-rcdfn-prime
 (lambda (a b) (int-f a b (n)))))))))
```

- Step 4: Prove the main theorem by functionally instantiating the zero-arity function(s) in the lemma introduced in step 3 with the corresponding free argument(s).

```
(defthm f-ftc-2
  (implies (and (integerp n) ;; we assume the type of n is integer.
                (inside-interval-p a (g-domain))
                (inside-interval-p b (g-domain)))
           (equal (int-f a b n)
                  (- (g b n)
                     (g a n))))
  :hints (("Goal"
           :by (:functional-instance f-ftc-2-lemma
                                     (n (lambda ()
                                          (if (integerp n) n 0)))))))
```

5 Orthogonality Relations of Trigonometric Functions

By applying the FTC-2 evaluation procedure, we can mechanically prove in ACL2(r) the orthogonality relations of trigonometric functions, which are the essential properties in Fourier series analysis. The orthogonality relations of trigonometric functions are a collection of definite integral formulas for sine and cosine functions as described below:

$$\int_{-L}^{L} \sin(m\frac{\pi}{L}x)\sin(n\frac{\pi}{L}x)dx = \begin{cases} 0, \text{ if } m \neq n \vee m = n = 0 \\ L, \text{ if } m = n \neq 0 \end{cases} \tag{5.1}$$

$$\int_{-L}^{L} \cos(m\frac{\pi}{L}x)\cos(n\frac{\pi}{L}x)dx = \begin{cases} 0, \text{ if } m \neq n \\ L, \text{ if } m = n \neq 0 \\ 2L, \text{ if } m = n = 0 \end{cases} \tag{5.2}$$

$$\int_{-L}^{L} \sin(m\frac{\pi}{L}x)\cos(n\frac{\pi}{L}x)dx = 0 \tag{5.3}$$

where $x, L \in \mathbb{R}$; $L \neq 0$; and $m, n \in \mathbb{N}$. As mentioned, these integral formulas can be proven using the FTC-2 evaluation procedure. Let's consider the case $m \neq n$ in formula (5.1); the other cases can be proven similarly. When $m \neq n$, formula (5.1) states that $\int_{-L}^{L} f(x,m,n,L)dx = 0$ where $f(x,m,n,L) = \sin(m\frac{\pi}{L}x)\sin(n\frac{\pi}{L}x)$. Using the automatic differentiator, we can easily prove that the function g defined below is indeed an antiderivative of f when $m \neq n$:

$$g(x,m,n,L) = \frac{1}{2}\left(\frac{\sin\left((m-n)\frac{\pi}{L}x\right)}{(m-n)\frac{\pi}{L}} - \frac{\sin\left((m+n)\frac{\pi}{L}x\right)}{(m+n)\frac{\pi}{L}}\right)$$

Then, by the FTC-2 theorem,

$$\begin{aligned}\int_{-L}^{L} f(x,m,n,L)dx &= g(L,m,n,L) - g(-L,m,n,L)\\ &= \frac{1}{2}\left(\frac{\sin((m-n)\pi)}{(m-n)\frac{\pi}{L}} - \frac{\sin((m+n)\pi)}{(m+n)\frac{\pi}{L}}\right) - \frac{1}{2}\left(\frac{\sin(-(m-n)\pi)}{(m-n)\frac{\pi}{L}} - \frac{\sin(-(m+n)\pi)}{(m+n)\frac{\pi}{L}}\right)\\ &= \frac{1}{2}(0-0) - \frac{1}{2}(0-0) = 0\end{aligned}$$

6 Sum Rule for Definite Integrals of Indexed Sums

As part of the Fourier coefficient formalization, we need to formalize the sum rule for definite integrals of indexed sums, which is stated as the following theorem:

Theorem 2 (Sum rule for definite integrals of indexed sums)**.** *Let $\{f_n\}$ be a set of real-valued continuous functions on $[a,b]$, where $n = 0,1,2,...,N$. Then*

$$\int_a^b \sum_{n=0}^{N} f_n(x)dx = \sum_{n=0}^{N} \int_a^b f_n(x)dx$$

Note that $f_n(x)$ abbreviates $f(x,n)$, which contains a free argument, n. Kaufmann [11] formalized the FTC-1 theorem for generic unary functions as a non-classical theorem. We re-prove it for generic functions with an extra argument added, following the method for extending the limited property of Riemann sums as described in Section 4.2. Then, by applying the FTC-2 evaluation procedure along with the extended version of FTC-1 and the sum rule for differentiation, the above theorem can be proven as follows:

Proof. For all $x \in [a,b]$ and $n = 0,1,\ldots,N$, let

$$g_n(x) = \int_a^x f_n(t)dt.$$

By FTC-1, $g_n'(x) = f_n(x)$ for all $x \in [a,b], n = 0,1,...,N$.

By the sum rule for differentiation, $\left(\sum_{n=0}^{N} g_n(x)\right)' = \sum_{n=0}^{N} g_n'(x) = \sum_{n=0}^{N} f_n(x)$ for all $x \in [a,b]$. Then, by FTC-2,

$$\begin{aligned}\int_a^b \sum_{n=0}^{N} f_n(x)dx &= \sum_{n=0}^{N} g_n(b) - \sum_{n=0}^{N} g_n(a)\\ &= \sum_{n=0}^{N} \int_a^b f_n(t)dt - \sum_{n=0}^{N} \int_a^a f_n(t)dt = \sum_{n=0}^{N} \int_a^b f_n(x)dx\end{aligned}$$

□

7 Fourier Coefficient Formulas

From the orthogonality relations and the sum rule for integration, the Fourier coefficients of periodic functions can be stated as follows:

Theorem 3 (Fourier coefficient formulas)**.** *Consider the following Fourier sum $f(x)$ for a periodic function with period $2L$:*

$$f(x) = a_0 + \sum_{n=1}^{N} \left(a_n \cos(n\frac{\pi}{L}x) + b_n \sin(n\frac{\pi}{L}x) \right) \tag{7.1}$$

Then

$$a_0 = \frac{1}{2L} \int_{-L}^{L} f(x)dx, \tag{7.2}$$

$$a_n = \frac{1}{L} \int_{-L}^{L} f(x) \cos(n\frac{\pi}{L}x)dx, \tag{7.3}$$

$$b_n = \frac{1}{L} \int_{-L}^{L} f(x) \sin(n\frac{\pi}{L}x)dx. \tag{7.4}$$

The proof of this theorem is straightforward from the orthogonality relations and the sum rule for integration, after applying the definition of f in (7.1) to the Fourier coefficient formulas (7.2), (7.3), and (7.4). Consequently, we can easily derive the following corollary. This is known as the uniqueness of Fourier sums:

Corollary 1 (Uniqueness of Fourier sums)**.** *Let*

$$f(x) = a_0 + \sum_{n=1}^{N} \left(a_n \cos(n\frac{\pi}{L}x) + b_n \sin(n\frac{\pi}{L}x) \right)$$

and

$$g(x) = A_0 + \sum_{n=1}^{N} \left(A_n \cos(n\frac{\pi}{L}x) + B_n \sin(n\frac{\pi}{L}x) \right)$$

$$\textit{Then } f = g \Leftrightarrow \begin{cases} a_0 = A_0 \\ a_n = A_n, \textit{ for all } n = 1,2,...,N \\ b_n = B_n, \textit{ for all } n = 1,2,...,N \end{cases}$$

Proof. ($\Rightarrow$) Follows immediately from the Fourier coefficient formulas:

$$a_0 = \frac{1}{2L} \int_{-L}^{L} f(x)dx = \frac{1}{2L} \int_{-L}^{L} g(x)dx = A_0,$$

$$a_n = \frac{1}{L} \int_{-L}^{L} f(x) \cos(n\frac{\pi}{L}x)dx = \frac{1}{L} \int_{-L}^{L} g(x) \cos(n\frac{\pi}{L}x)dx = A_n,$$

$$b_n = \frac{1}{L} \int_{-L}^{L} f(x) \sin(n\frac{\pi}{L}x)dx = \frac{1}{L} \int_{-L}^{L} g(x) \sin(n\frac{\pi}{L}x)dx = B_n.$$

($\Leftarrow$) Obviously true by induction on n. □

8 Sum Rule for Definite Integrals of Infinite Series

Our formalization of the sum rule for integration in Section 6 only applies to finite sums. However, Fourier series can be infinite. Thus, the sum rule for integration needs to be extended to infinite series; we do so in this section.

Our basic result is the following sum rule for integrals of infinite series, formalized using non-standard analysis. (We define uniform convergence below.)

Theorem 4. *Suppose that $\{f_n(x)\}$ is a sequence of real-valued continuous functions whose sequence of partial sums converges uniformly to a continuous limit function on a given interval. Then*

$$\int_a^b st\left(\sum_{n=0}^{H_0} f_n(x)\right) dx = st\left(\sum_{n=0}^{H_1} \int_a^b f_n(x)dx\right) \tag{8.1}$$

for all infinitely large natural numbers H_0 and H_1.

Remark. The conclusion above is equivalent to the following formula using the *epsilon-delta* definition of limit [14]:

$$\int_a^b \lim_{N\to\infty}\left(\sum_{n=0}^{N} f_n(x)\right) dx = \lim_{N\to\infty}\left(\sum_{n=0}^{N} \int_a^b f_n(x)dx\right) \tag{8.2}$$

We turn now to two variants of this theorem that relax its requirements. We start by recalling well-known formulations of convergence in non-standard analysis.

- **Pointwise convergence:** Suppose $\{f_n\}$ is a sequence of functions defined on an interval I. The sequence $\{f_n\}$ converges *pointwise* to the *limit function* f on the interval I if $f_H(x) \approx f(x)$ for all *standard* $x \in I$ and for all infinitely large natural numbers H.
- **Uniform convergence:** Suppose $\{f_n\}$ is a sequence of functions defined on an interval I. The sequence $\{f_n\}$ converges *uniformly* to the *limit function* f on the interval I if $f_H(x) \approx f(x)$ for all $x \in I$ (both *standard* and *non-standard*) and for all infinitely large natural numbers H.

Clearly, uniform convergence is stronger than pointwise convergence. A sequence that converges uniformly to a limit function also converges pointwise to that function, but the reverse is not guaranteed. We meet the hypothesis of Theorem 4 — uniform convergence to a continuous limit function — in two ways corresponding to two different conditions, as follows. Note: Only the second condition is relevant to Fourier series, but the first also leads to an interesting result.

- **Condition 1:** A *monotone* sequence of partial sums of real-valued continuous functions *converges pointwise* to a *continuous* limit function on the *closed and bounded* interval of interest.
- **Condition 2:** A sequence of partial sums of real-valued continuous functions *converges uniformly* to a limit function on the interval of interest.

The following theorem [15] shows that Condition 1 implies the hypothesis of Theorem 4.

Theorem 5 (Dini Uniform Convergence Theorem)**.** *A monotone sequence of continuous functions $\{f_n\}$ that converges pointwise to a continuous function f on a closed and bounded interval $[a,b]$ is uniformly convergent.*

Our proof of Dini's theorem relies on the *overspill principle* from non-standard analysis [9, 13]. Thus, we now discuss our formalization of this principle in ACL2(r) [3].

Overspill principle (weak version): Let $P(n,x)$ be a classical predicate. Then

$$\forall x.\left(\left(\forall^{st} n \in \mathbb{N}.P(n,x)\right) \Rightarrow \exists^{\neg st} k \in \mathbb{N}.P(k,x)\right). \tag{8.3}$$

In words, if a classical predicate P holds for all standard natural numbers n, P must be hold for some non-standard natural number k. By applying this principle, we can even come up with a stronger statement as follows:

Overspill principle (strong version): Let $P(n,x)$ be a classical predicate. Then

$$\forall x.\left(\left(\forall^{st} n \in \mathbb{N}.P(n,x)\right) \Rightarrow \exists^{\neg st} k \in \mathbb{N}.\forall m \in \mathbb{N}.\left(m \leq k \Rightarrow P(m,x)\right)\right). \tag{8.4}$$

In words, if a classical predicate P holds for all standard natural numbers n, there must exist some non-standard natural number k such that P holds for all natural numbers less than or equal to k. (8.4) can be derived from (8.3) through a classical predicate $P^*(n,x)$ defined in terms of $P(n,x)$ as follows:

```
(defun P* (n x)
  (if (zp n)
      (P 0 x)
    (and (P n x)
         (P* (1- n) x))))
```

The proof of (8.4) now proceeds as follows.

1. Fix x and assume $(\forall^{st} n \in \mathbb{N}.P(n,x))$.
2. Then, we can show that $(\forall^{st} m \in \mathbb{N}.P^*(m,x))$.
3. Applying (8.3) to P^*: $\exists^{\neg st} k \in \mathbb{N}.P^*(k,x)$.
4. From the definition of P^*: $\exists^{\neg st} k \in \mathbb{N}.\forall m \in \mathbb{N}.\left(m \leq k \Rightarrow P(m,x)\right)$.

We formalize (8.4) for a generic classical predicate $P(n,x)$ in ACL2(r) and provide the `overspill` utility, which automates the application of (8.4). In particular, the user needs only to define a classical predicate P_1 and then call the `overspill` macro with the input P_1 so that (8.4) will be applied to P_1 automatically via a functional instantiation. We can thus apply the overspill principle (8.4) to prove Dini's theorem as shown below.

Proof of Theorem 5. Without loss of generality, assume $\{f_n\}$ is *monotonically increasing*. We want to prove $f(x) \approx f_H(x)$ for all $x \in [a,b]$ and for all infinitely large $H \in \mathbb{N}$.

Fact: If $x \in [a,b]$ then $\mathrm{st}(x) \in [a,b]$ (note that this is only true on *closed and bounded intervals*).

(A) Since $\mathrm{st}(x)$ is standard and $x \approx \mathrm{st}(x)$, $f(x) \approx f(\mathrm{st}(x))$ by the continuity of f.

(B) Since $\mathrm{st}(x)$ is standard, $f(\mathrm{st}(x)) \approx f_H(\mathrm{st}(x))$ by the pointwise convergence of $\{f_n\}$.

We will make the following two claims.

Claim 1. For some non-standard $k \in \mathbb{N}$, we have: for all $H \leq k$, $f_H(\mathrm{st}(x)) \approx f_H(x)$.

Claim 2. Suppose $k \in \mathbb{N}$ such that $f_k(x) \approx f(x)$. Then for all $H > k$, $f(x) \approx f_H(x)$.

For the moment, assume both claims. Choosing k according to Claim 1, then by the transitivity of `i-close` and Steps (A) and (B) above, we have $f(x) \approx f_H(x)$ for all infinitely large $H \leq k$. Applying Claim 2 to that same k takes care of $H > k$, so we are done once we prove the claims.

To prove Claim 1, we must find a non-standard k such that $f_H(\mathrm{st}(x)) \approx f_H(x)$ for all $H \leq k$. We first observe that by the continuity of $\{f_n\}$, we have $f_n(\mathrm{st}(x)) \approx f_n(x), \forall x \in [a,b]$ and $\forall^{st} n \in \mathbb{N}$. We apply the overspill principle — which requires a classical predicate — to the following classical predicate $P(n,x_0,x)$.

$$P(n,x_0,x) \equiv |f_n(x_0) - f_n(x)| < \frac{1}{n+1}$$

If $x_0, x \in [a,b]$, x_0 is standard, and $x_0 \approx x$, then $P(n,x_0,x)$ holds for all *standard* $n \in \mathbb{N}$ since $f_n(x_0) - f_n(x) \approx 0$ by the continuity of $\{f_n\}$. Hence, by the overspill principle (8.4), there exists a *non-standard* $k \in \mathbb{N}$ s.t. $P(m,x_0,x)$ holds for all $m \in \mathbb{N}$ and $m \leq k$. Now suppose that H is infinitely large but $H \leq k$. Then $f_H(x_0) \approx f_H(x)$ since

$$0 \leq |f_H(x_0) - f_H(x)| < \frac{1}{H+1} \approx 0.$$

Let's pick x_0 to be st(x); then $f_H(\text{st}(x)) \approx f_H(x)$, concluding the proof of Claim 1.

To prove Claim 2, by hypothesis pick $k \in \mathbb{N}$ such that $f_k(x) \approx f(x)$, and assume $H > k$. Then $f_k(x) \leq f_H(x) \leq f(x)$ by the increasing monotonicity of $\{f_n\}$. Hence, $0 \leq |f(x) - f_H(x)| \leq |f(x) - f_k(x)| \approx 0$. Thus, $f(x) \approx f_H(x)$, which concludes the proof of Claim 2 and also the proof of Theorem 5. □

Dini's theorem shows that pointwise convergence of a sequence of continuous functions on a closed and bounded interval also implies its uniform convergence if the sequence is monotonic and the limit function is continuous. Unfortunately, it is not applicable to Fourier series since Fourier series are not required to be monotonic. As a result, Fourier series cannot meet the requirement for our proof of (4) from Condition 1. In fact, Fourier series can satisfy Condition 2 under suitable criteria [10]. Then, from Condition 2, we need to prove that the limit function is continuous in order to meet the requirement for our proof of (4). This can be proven by applying the overspill principle.

Theorem 6. *Suppose that a sequence of continuous functions $\{f_n\}$ converges uniformly to a limit function f on an interval I. Then f is also continuous on I.*

Proof. The goal is to prove $f(x_0) \approx f(x)$ for all $x_0, x \in I$ such that x_0 is standard and $x_0 \approx x$. By the uniform convergence of $\{f_n\}$, we have $f(x_0) \approx f_H(x_0)$ and $f(x) \approx f_H(x)$ for all infinitely large $H \in \mathbb{N}$. If we can show that $f_H(x_0) \approx f_H(x)$, then we obtain our goal by the transitivity of $\approx$. By applying the overspill principle in the same way as in our proof of Theorem 5, we claim that there must exist a non-standard $k \in \mathbb{N}$ s.t. $f_H(x_0) \approx f_H(x)$ if $H \leq k$. When $H > k$, we know that $f_H(x_0) \approx f_k(x_0)$ (since they are both `i-close` to $f(x_0)$ by the uniform convergence of $\{f_n\}$) and similarly $f_H(x) \approx f_k(x)$. Thus, $f_H(x_0) \approx f_k(x_0) \approx f_k(x) \approx f_H(x)$ and we are done. □

From Condition 2 and Theorem 6, in order to apply the sum rule for integration (4) to infinite Fourier series, we need to prove that the Fourier series converge uniformly to limit functions. As mentioned above, this is provable under suitable criteria [10].

9 Conclusions

We described in this paper our extension of the framework for formally evaluating definite integrals of real-valued continuous functions containing free arguments, using FTC-2. Along with this extension, we also presented our technique for handling the occurrence of free arguments in pseudo-lambda expressions of functional instantiations. Using the extended framework, we showed how to prove the orthogonality relations of trigonometric functions as well as the sum rule for definite integrals of indexed sums. These properties were then applied to prove the Fourier coefficient formulas and consequently used to derive the uniqueness of Fourier sums as a corollary.

We also presented our formalization of the sum rule for definite integrals of infinite series under two different conditions. Along with this task, we formalized the overspill principle and provided the `overspill` utility that automates the application of the overspill principle, thus strengthening the reasoning capability of non-standard analysis in ACL2(r). Our proofs of Dini's theorem and the continuity of the limit function as described in Section 8 illustrate this capability.

Some possible areas of future work are worth mentioning. First, the automatic differentiator needs to be extended to support partial differentiation. The current AD has limited support for automating

partial differentiation. Although we extended the AD to support partial derivative registrations of binary functions, this extension is still very limited for automatic differentiation. In particular, our extension imposes a constraint on the free argument of binary functions that either its symbolic name must be *arg*0 or it has to be a constant. As a result, the AD cannot be applied to expressions containing several binary functions with different free arguments. Our current solution in this case is to break those expressions into smaller expressions such that the AD can be applied directly to these smaller expressions, and then manually combine them to get the final result for the original expressions. Future work might make the partial differentiation process more automatic. Another possibility for future work is to prove convergence of the Fourier series for a periodic function, under sufficient conditions.

In summary, we have developed and extended frameworks for mechanized continuous mathematics, which we applied to obtain results about Fourier series and an elegant proof of Dini's theorem. We are confident that our frameworks can be applied to future work on Fourier series and, more generally, continuous mathematics, to be carried out in ACL2(r).

Acknowledgements

We thank Ruben Gamboa for useful discussions. We also thank the reviewers for useful comments. This material is based upon work supported by DARPA under Contract No. N66001-10-2-4087.

References

[1] ACL2: *ACL2 Documentation on Lemma-Instance*. See URL `http://www.cs.utexas.edu/users/moore/acl2/current/manual/index.html?topic=ACL2____LEMMA-INSTANCE`.

[2] ACL2 Community Books: Available at `https://github.com/acl2/acl2`.

[3] Overspill Principle Formalization Source Code: Available at `https://raw.githubusercontent.com/acl2/acl2/master/books/nonstd/nsa/overspill.lisp`.

[4] R. Gamboa & J. Cowles (2007): *Theory Extension in ACL2(r)*. *Journal of Automated Reasoning* 38(4), pp. 273–301, doi:10.1007/s10817-006-9043-0.

[5] J. Cowles & R. Gamboa (2014): *Equivalence of the Traditional and Non-Standard Definitions of Concepts from Real Analysis*. In: *Proc of the Twelfth International Workshop on the ACL2 Theorem Prover and its Applications (ACL2-2014)*, pp. 89–100, doi:10.4204/EPTCS.152.8.

[6] P. Reid & R. Gamboa (2011): *Automatic Differentiation in ACL2*. In: *Proc of the Second International Conference on Interactive Theorem Proving (ITP-2011)*, pp. 312–324, doi:10.1007/978-3-642-22863-6_23.

[7] P. Reid & R. Gamboa (2011): *Implementing an Automatic Differentiator in ACL2*. In: *Proc of the Tenth International Workshop on the ACL2 Theorem Prover and its Applications (ACL2-2011)*, pp. 61–69, doi:10.4204/EPTCS.70.5.

[8] R. Gamboa (1999): *Mechanically Verifying Real-Valued Algorithms in ACL2*. Ph.D. thesis, The University of Texas at Austin.

[9] R. Goldblatt (1998): *Lectures on the Hyperreals: An Introduction to Nonstandard Analysis*. Springer.

[10] D. Jackson (1934): *The Convergence of Fourier Series*. *The American Mathematical Monthly* 41(2), pp. 67–84, doi:10.2307/2300327.

[11] M. Kaufmann (2000): *Modular Proof: The Fundamental Theorem of Calculus*. In M. Kaufmann, P. Manolios & J S. Moore, editors: *Computer-Aided Reasoning: ACL2 Case Studies*, chapter 6, 4, Springer US, pp. 75–91, doi:10.1007/978-1-4757-3188-0_6.

[12] R. Gamboa & M. Kaufmann (2001): *Non-Standard Analysis in ACL2*. *Journal of Automated Reasoning* 27(4), pp. 323–351, doi:10.1023/A:1011908113514.

[13] H. J. Keisler (1976): *Foundations of Infinitesimal Calculus*. Prindle Weber & Schmidt.

[14] H. J. Keisler (1985): *Elementary Calculus: An Infinitesimal Approach*. Prindle Weber & Schmidt.

[15] W. A. J. Luxemburg (1971): *Arzela's Dominated Convergence Theorem for the Riemann Integral*. The American Mathematical Monthly 78(9), pp. 970–979, doi:10.2307/2317801.

[16] Inc. Wolfram Research (2015): *Mathematica*. Available at `http://www.wolfram.com/mathematica/`.

Perfect Numbers in ACL2

John Cowles Ruben Gamboa

Department of Computer Science
University of Wyoming
Laramie, Wyoming, USA

cowles@uwyo.edu ruben@uwyo.edu

A **perfect number** is a positive integer n such that n equals the sum of all positive integer divisors of n that are less than n. That is, although n is a divisor of n, n is excluded from this sum. Thus $6 = 1+2+3$ is perfect, but $12 \neq 1+2+3+4+6$ is not perfect. An ACL2 theory of perfect numbers is developed and used to prove, in ACL2(r), this bit of mathematical folklore: Even if there are infinitely many perfect numbers, the series below, of the reciprocals of all perfect numbers, converges.

$$\sum_{\text{perfect } n} \frac{1}{n}$$

1 Perfect Numbers

The smallest perfect numbers are $6 = 2 \cdot 3 = 2^1(2^2-1)$, $28 = 4 \cdot 7 = 2^2(2^3-1)$, $496 = 16 \cdot 31 = 2^4(2^5-1)$, $8128 = 64 \cdot 127 = 2^6(2^7-1)$. In each of these examples, the second factor, $3, 7, 31, 127$, of the form $2^k - 1$, is a prime. The Greek Euclid proved [2, page 3]:

Theorem 1 *If $2^k - 1$ is prime, then $n = 2^{k-1}(2^k-1)$ is perfect.*

Primes of the form $2^k - 1$ are called **Mersenne primes**. Thus every new Mersenne prime leads to a new perfect number. According to Wikipedia [5], less than 50 Mersenne primes are known. The largest known Mersenne prime is $2^{57,885,161} - 1$, making $2^{57,885,160}(2^{57,885,161} - 1)$ the largest known perfect number, with over 34 million digits. It is not known if there are infinitely many Mersenne primes, nor if there are infinitely many perfect numbers.

All perfect numbers built from Mersenne primes are even. The Swiss Euler proved every **even** perfect number is built from some Mersenne prime [2, page 10]:

Theorem 2 *If n is an even perfect number, then $n = 2^{k-1}(2^k-1)$, where $2^k - 1$ is prime.*

It is not known if there are any odd perfect numbers, but Euler also proved [6, page 250]:

Theorem 3 *If n is an odd perfect number, then $n = p^i m^2$, where p is prime and i, p, m are odd.*

ACL2 is used to verify each of these three theorems.

If there are only finitely many perfect numbers, then clearly the series

$$\sum_{\text{perfect } n} \frac{1}{n}$$

converges. ACL2(r) is used to verify that even if there are **infinitely many perfect numbers**, the series converges.

M. Kaufmann and D. Rager (Eds.): ACL2 Workshop 2015 (ACL2 2015).
EPTCS 192, 2015, pp. 53–59, doi:10.4204/EPTCS.192.5

2 The ACL2 Theory

In number theory, for positive integer n, $\sigma(n)$ denotes the sum of **all** (including n) positive integer divisors of n. The function $\sigma(n)$ has many useful properties, so the definition of a perfect number is reformulated in terms of σ [2, pages 8–9]:

$$\text{perfect}(n) \text{ if and only if } \sigma(n) = 2n$$

.

These six properties of σ are among those formulated and proved in ACL2:

1. p is prime if and only if $\sigma(p) = p+1$.
2. If p is prime, then $\sigma(p^k) = \sum_{i=0}^{k} p^i = \frac{p^{k+1}-1}{p-1}$.
3. If p and q are different primes, then $\sigma(p \cdot q) = \sigma(p) \cdot \sigma(q)$.
4. $\sigma(k \cdot n) \leq \sigma(k) \cdot \sigma(n)$
5. If $\gcd(k,n) = 1$, then $\sigma(k \cdot n) = \sigma(k) \cdot \sigma(n)$.
6. If p is prime, then $\gcd(p^k, \sigma(p^k)) = 1$.

If $n = 2^i(2^{i+1}-1)$ is an even perfect number, then the exponent i is computed by an ACL2 term, `(cdr (odd-2^i n))`, that returns the largest value of i such that 2^i divides n.

If $n = p^i m^2$ is an odd perfect number, then p, i, m are respectively computed by the ACL2 terms

- ```
  (car (find-pair-with-odd-cdr
         (prime-power-factors n)))
  ```

- ```
  (cdr (find-pair-with-odd-cdr
         (prime-power-factors n)))
  ```

- ```
  (product-pair-lst (pairlis$ (strip-cars
                               (remove1-equal
                                (find-pair-with-odd-cdr
                                 (prime-power-factors n))
                                (prime-power-factors n)))
                              (map-nbr-product
                               1/2
                               (strip-cdrs
                                (remove1-equal
                                 (find-pair-with-odd-cdr
                                  (prime-power-factors n))
                                 (prime-power-factors n))))))
  ```

These terms implement the following computation:

1. Factor $n = \prod_{j=0}^{k} p_j^{e_j}$ into the product of powers of distinct odd primes.
2. Exactly one of the exponents, say e_0, will be odd and all the other exponents will be even.
3. p is the prime with the odd exponent and i is the unique odd exponent. So $n = p^i \cdot \prod_{j=1}^{k} p_j^{2f_j}$.
  ```

4. Then $m = \prod_{j=1}^{k} p_j^{f_j}$ and $n = p^i m^2$.

ACL2 is used to verify a result of B. Hornfeck, that different odd perfect numbers, $n_1 = p_1^{i_1} m_1^2 \neq n_2 = p_2^{i_2} m_2^2$ have distinct m_i [6, page 251]:

Theorem 4 *If $n_1 = p_1^{i_1} m_1^2$ and $n_2 = p_2^{i_2} m_2^2$ are odd perfect numbers and $m_1 = m_2$, then $n_1 = n_2$.*

Theorems 2, 3, and 4 are enough to prove the folklore that the series, of the reciprocals of all perfect numbers, converges.

3 ACL2(r)

ACL2(r) [3] is based on **Nonstandard Analysis** [7, 4] which provides rigorous foundations for reasoning about real, complex, infinitesimal, and infinite quantities. There are two versions of the **reals**

1. The **Standard Reals**, ${}^{\mathrm{st}}\mathbb{R}$, is the unique **complete** ordered field. This means that
   - Every nonempty subset of ${}^{\mathrm{st}}\mathbb{R}$ that is bounded above has a **least upper bound**.

   There are no non-zero infinitesimal elements, nor are there are any infinite elements in ${}^{\mathrm{st}}\mathbb{R}$.
2. The **HyperReals**, ${}^{\star}\mathbb{R}$, is a proper field extension of ${}^{\mathrm{st}}\mathbb{R}$: ${}^{\mathrm{st}}\mathbb{R} \subsetneq {}^{\star}\mathbb{R}$. There are non-zero infinitesimal elements and also infinite elements in ${}^{\star}\mathbb{R}$.

Here are some technical definitions.

- $x \in {}^{\star}\mathbb{R}$ is **infinitesimal**: For all positive $r \in {}^{\mathrm{st}}\mathbb{R}$, $(|x| < r)$.
  0 is the only infinitesimal in ${}^{\mathrm{st}}\mathbb{R}$.
  `(i-small x)` in ACL2(r).
- $x \in {}^{\star}\mathbb{R}$ is **finite**: For some $r \in {}^{\mathrm{st}}\mathbb{R}$, $(|x| < r)$.
  `(i-limited x)` in ACL2(r).
- $x \in {}^{\star}\mathbb{R}$ is **infinite**: For all $r \in {}^{\mathrm{st}}\mathbb{R}$, $(|x| > r)$.
  `(i-large x)` in ACL2(r)
- $x, y \in {}^{\star}\mathbb{R}$ are **infinitely close**, $x \approx y$: $x - y$ is infinitesimal.
  `(i-close x y)` in ACL2(r).
- $\mathfrak{n}_\infty$ is an infinite positive integer constant.
  `(i-large-integer)` in ACL2(r).

Every (partial) function $f : {}^{\mathrm{st}}\mathbb{R}^n \longmapsto {}^{\mathrm{st}}\mathbb{R}^k$ has an extension ${}^{\star}f : {}^{\star}\mathbb{R}^n \longmapsto {}^{\star}\mathbb{R}^k$ such that

1. For $x_1, \cdots, x_n \in {}^{\mathrm{st}}\mathbb{R}$, ${}^{\star}f(x_1, \cdots, x_n) = f(x_1, \cdots, x_n)$.
2. Every first-order statement about f true in ${}^{\mathrm{st}}\mathbb{R}$ is true about ${}^{\star}f$ in ${}^{\star}\mathbb{R}$.

   Example.
   $(\forall x)[\sin^2(x) + \cos^2(x) = 1]$ is true in ${}^{\mathrm{st}}\mathbb{R}$.
   $(\forall x)[{}^{\star}\sin^2(x) + {}^{\star}\cos^2(x) = 1]$ is true in ${}^{\star}\mathbb{R}$.

Any (partial) function $f : {}^{\mathrm{st}}\mathbb{R}^n \longmapsto {}^{\mathrm{st}}\mathbb{R}^k$ is said to be **classical**.

- Identify a classical f with its extension ${}^{\star}f$.
  That is, use f for both the original classical function f and its extension ${}^{\star}f$.

- Use $(\forall^{\mathrm{st}} x)$ for $(\forall x \in {}^{\mathrm{st}}\mathbb{R})$, i.e. "for all **standard** x."
  Use $(\exists^{\mathrm{st}} x)$ for $(\exists x \in {}^{\mathrm{st}}\mathbb{R})$, i.e. "there is some **standard** x."
- "$(\forall x)[\sin^2(x)+\cos^2(x)=1]$ is true in ${}^{\mathrm{st}}\mathbb{R}$" becomes "$(\forall^{\mathrm{st}} x)[\sin^2(x)+\cos^2(x)=1]$ is true in ${}^{\star}\mathbb{R}$."
  "$(\forall x)[{}^{\star}\sin^2(x)+{}^{\star}\cos^2(x)=1]$ is true in ${}^{\star}\mathbb{R}$" becomes "$(\forall x)[\sin^2(x)+\cos^2(x)=1]$ is true in ${}^{\star}\mathbb{R}$."

Numeric constants, c, are viewed as 0-ary functions, $c : {}^{\mathrm{st}}\mathbb{R}^0 \longmapsto {}^{\mathrm{st}}\mathbb{R}$ or $c : {}^{\star}\mathbb{R}^0 \longmapsto {}^{\star}\mathbb{R}$. Thus, elements of ${}^{\mathrm{st}}\mathbb{R}$, such as $2, 4, -1$, are classical. But elements of ${}^{\star}\mathbb{R} - {}^{\mathrm{st}}\mathbb{R}$, such as the infinite positive integer $\mathfrak{n}_\infty$, are not classical. Functions defined using the nonstandard concepts of infinitesimal, finite, infinite, and infinitely close are not classical.

Let f be a (partial) unary function, whose domain includes the **nonnegative** integers, into the reals. Here are three possible definitions for the real series $\sum_{i=0}^{\infty} f(i)$ converges. The first two are versions of Weierstrass' traditional definition that the real series converges. One version for the standard reals, another version for the hyperreals.

1. `(defun-sk Series-Converges-Traditional-Standard ( )`
   $(\exists^{\mathrm{st}} L)(\forall^{\mathrm{st}} \varepsilon > 0)(\exists^{\mathrm{st}}\ \text{integer}\ M > 0)(\forall^{\mathrm{st}}\ \text{integer}\ n)(n > M \Rightarrow |\sum_{i=0}^{n} f(i) - L| < \varepsilon)$
   `)`
2. `(defun-sk Series-Converges-Traditional-Hyper ( )`
   $(\exists L)(\forall \varepsilon > 0)(\exists\ \text{integer}\ M > 0)(\forall\ \text{integer}\ n)(n > M \Rightarrow |\sum_{i=0}^{n} f(i) - L| < \varepsilon)$
   `)`
3. `(defun-sk Series-Converges-Infinitesimal ( )`
   $(\exists^{\mathrm{st}} L)(\forall\ \text{infinite integer}\ n > 0)(\sum_{i=0}^{n} f(i) \approx L)$
   `)`

For **classical** f, ACL2(r) verifies these three definitions are equivalent. ACL2(r) also verifies for classical f, with **nonnegative** range, these definitions are equivalent to this nonstandard definition [1]:

- `(defun Series-Converges-Nonstandard ( )`
  $\sum_{i=0}^{\mathfrak{n}_\infty} f(i)$ is finite
  `)`
  Recall that the upper limit, $\mathfrak{n}_\infty$, on this $\sum_{i=0}^{\mathfrak{n}_\infty} f(i)$, is an infinite positive integer constant.

4 The Series Converges

Use the definition, `Series-Converges-Nonstandard`, to verify, in ACL2(r), the convergence of

$$\sum_{\mathrm{perfect}(k)} \frac{1}{k} = \sum_{\substack{k=1 \\ \mathrm{perfect}(k)}}^{\infty} \frac{1}{k}$$

by showing this sum is finite:

$$\sum_{\substack{k=1 \\ \mathrm{perfect}(k)}}^{\mathfrak{n}_\infty} \frac{1}{k}$$

Recall $\mathfrak{n}_\infty$ is an infinite positive integer constant.
Verify the previous sum is finite by showing both of the summands on the right side below are finite.

$$\sum_{\substack{k=1\\ \text{perfect}(k)}}^{\mathfrak{n}_\infty} \frac{1}{k} \;=\; \sum_{\substack{k=1\\ \text{perfect}(k)\\ \text{even}(k)}}^{\mathfrak{n}_\infty} \frac{1}{k} \;+\; \sum_{\substack{k=1\\ \text{perfect}(k)\\ \text{odd}(k)}}^{\mathfrak{n}_\infty} \frac{1}{k}$$

By Theorem 2, even perfect numbers, k, have the form $k = 2^i(2^{i+1}-1)$. Since $2^i(2^{i+1}-1) \geq 2^i$, $\frac{1}{2^i(2^{i+1}-1)} \leq \frac{1}{2^i}$. Induction on n verifies $\sum_{i=0}^{n} \frac{1}{2^i} = 2 - \frac{1}{2^n}$. Thus for any positive integer, n, including $n = \mathfrak{n}_\infty$:

$$0 \;\leq\; \sum_{\substack{k=1\\ \text{perfect}(k)\\ \text{even}(k)}}^{n} \frac{1}{k} \;=\; \sum_{\substack{k=1\\ \text{perfect}(k)\\ k=2^i(2^{i+1}-1)}}^{n} \frac{1}{2^i(2^{i+1}-1)} \;\leq\; \sum_{\substack{k=1\\ \text{perfect}(k)\\ k=2^i(2^{i+1}-1)}}^{n} \frac{1}{2^i} \;\leq\; \sum_{i=0}^{n} \frac{1}{2^i} \;=\; 2 - \frac{1}{2^n} \;<\; 2$$

By Theorem 3, odd perfect numbers, k, have the form $k = p^i m^2$. Since $p^i m^2 \geq m^2$, $\frac{1}{p^i m^2} \leq \frac{1}{m^2}$. By Theorem 4, no square, m^2, appears more than once in

$$\sum_{\substack{k=1\\ \text{perfect}(k)\\ k=p^i m^2}}^{n} \frac{1}{m^2}$$

Induction on n verifies $\sum_{m=1}^{n} \frac{1}{m^2} \leq 2 - \frac{1}{n}$, Thus for any positive integer, n, including $n = \mathfrak{n}_\infty$:

$$0 \;\leq\; \sum_{\substack{k=1\\ \text{perfect}(k)\\ \text{odd}(k)}}^{n} \frac{1}{k} \;=\; \sum_{\substack{k=1\\ \text{perfect}(k)\\ k=p^i m^2}}^{n} \frac{1}{p^i m^2} \;\leq\; \sum_{\substack{k=1\\ \text{perfect}(k)\\ k=p^i m^2}}^{n} \frac{1}{m^2} \;\leq\; \sum_{m=1}^{n} \frac{1}{m^2} \;\leq\; 2 - \frac{1}{n} \;<\; 2$$

Therefore, for any positive integer, n, including $n = \mathfrak{n}_\infty$:

$$0 \;\leq\; \sum_{\substack{k=1\\ \text{perfect}(k)}}^{n} \frac{1}{k} \;=\; \sum_{\substack{k=1\\ \text{perfect}(k)\\ \text{even}(k)}}^{n} \frac{1}{k} \;+\; \sum_{\substack{k=1\\ \text{perfect}(k)\\ \text{odd}(k)}}^{n} \frac{1}{k} \;<\; 2+2=4$$

and

$$\sum_{\substack{k=1\\ \text{perfect}(k)}}^{\mathfrak{n}_\infty} \frac{1}{k} \text{ is finite.}$$

The heart of this proof is that the partial sums

$$\sum_{\substack{k=1\\ \text{perfect}(k)}}^{n} \frac{1}{k}$$

are bounded above (by 4). This can be stated and carried out entirely in ACL2. The **Reals** and ACL2(r) are required to formally state and prove the series converges.

A ACL2(r) Books

A.1 prime-fac.lisp

Unique Prime Factorization Theorem for Positive Integers.
An ACL2 book as well as an ACL2(r) book.

A.2 perfect.lisp

Perfect Positive Integers.
An ACL2 book as well as an ACL2(r) book.
Over 500 events, incrementally built Summer 2013 – Spring 2015.

A.3 series1.lisp

The CLASSICAL series, Ser1, converges (to a STANDARD real L).

A.4 series1a.lisp

The CLASSICAL NONNEGATIVE series, Ser1a, converges (to a STANDARD real L).

A.5 sumlist-1.lisp

Some nice events from sumlist.lisp plus additional events.

A.6 sum-recip-e-perfect.lisp

The sum of the RECIPROCALS of the EVEN PERFECT positive integers converges.

A.7 sum-recip-o-perfect.lisp

The sum of the RECIPROCALS of the ODD PERFECT positive integers converges.

A.8 sum-recip-perfect.lisp

The sum of the RECIPROCALS of the PERFECT positive integers converges.

References

[1] John Cowles & Ruben Gamboa (2014): *Equivalence of the Traditional and Non-Standard Definitions of Concepts from Real Analysis*. In Freek Verbeek & Julien Schmaltz, editors: *Proceedings of the Twelfth International Workshop of the ACL2 Theorem Prover and its Applications (ACL2-2014)*, Vienna, Austria, pp. 89–100, doi:10.4204/EPTCS.152.8.

[2] William Dunham (1999): *Euler: The Master of Us All*. Mathematical Association of America.

[3] Ruben Gamboa (1999): *Mechanically Verifying Real-Valued Algorithms in ACL2*. Ph.D. thesis, University of Texas at Austin.

[4] Edward Nelson (1977): *Internal Set Theory: A New Approach to Nonstandard Analysis*. *Bulletin of the American Mathematical Society* 83, pp. 1165–1198, doi:10.1090/S0002-9904-1977-14398-X.

[5] (2015): *Perfect Number: From Wikipedia, the free encyclopedia*. Available at `en.wikipedia.org/wiki/Perfect_number`.

[6] Paul Pollack (2009): *Not Always Buried Deep: A Second Course in Elementary Number Theory*. American Mathmatical Society.

[7] Abraham Robinson (1966): *Non-Standard Analysis*. North-Holland Publishing Co.

Extending ACL2 with SMT Solvers

Yan Peng Mark Greenstreet
University of British Columbia
Vancouver, Canada
yanpeng,mrg@cs.ubc.ca

We present our extension of ACL2 with Satisfiability Modulo Theories (SMT) solvers using ACL2's trusted clause processor mechanism. We are particularly interested in the verification of physical systems including Analog and Mixed-Signal (AMS) designs. ACL2 offers strong induction abilities for reasoning about sequences and SMT complements deduction methods like ACL2 with fast nonlinear arithmetic solving procedures. While SAT solvers have been integrated into ACL2 in previous work, SMT methods raise new issues because of their support for a broader range of domains including real numbers and uninterpreted functions. This paper presents `Smtlink`, our clause processor for integrating SMT solvers into ACL2. We describe key design and implementation issues and describe our experience with its use.

1 Introduction

This paper presents `Smtlink`, a clause processor for using satisfiability modulo theory (SMT) solvers to discharge proof goals in ACL2. Prior work has [21, 23] incorporated SAT solving into ACL2, and Manolios and Srinivasan [16, 22] described an extension of ACL2 with the Yices SMT solver. Our work explores the use of SMT solvers for their decision procedures for linear and non-linear arithmetic which, to the best of our knowledge, has not been addressed in prior work.

Interactive theorem proving and SMT solving provide complementary strengths for verification. SMT solvers can automatically discharge proof obligations that would be tedious to handle with an interactive theorem prover alone. Conversely, theorem provers provide methods for proof by induction and proof structuring methods. While there has been some work on automatically proving induction proofs using SMT solvers (see [15]), theorem provers such as ACL2 offer a much more comprehensive framework for induction proofs. For many problems, SMT solvers cannot prove the main result in a single step; in fact, the main theorem may not even be expressible in the logic of the SMT solver. However, the SMT solver can discharge key lemmas to simplify the proof process, and the theorem prover can ensure that the proofs for the main theorems are, indeed, complete. When used from within an interactive theorem prover, the user can identify key goals and *relevant* facts to make effective use of the SMT solver. Doing so can avoid sending the SMT solver down a path of an intractable number of useless branches and lead instead to a proof of the desired goal.

Our intended application of the combination of ACL2 with an SMT solver is to verify properties of Analog and Mixed-Signal (AMS) circuits and other cyber-physical systems. AMS circuits are mixed analog and digital systems, typically consisting of multiple analog and digital feedback loops operating at much different time scales. It is not practical to simulate AMS circuits for all possible device parameters, initial conditions, inputs, and operating conditions. In fact, running just one such simulation may require more time than the design schedule. Most AMS circuits are intended to be correct for relatively simple reasons - errors occur because the designer's informal reasoning overlooked some critical case or had some simple error. Our approach is to verify that the intuitive argument for correctness is indeed correct

M. Kaufmann and D. Rager (Eds.): ACL2 Workshop 2015 (ACL2 2015).
EPTCS 192, 2015, pp. 61–77, doi:10.4204/EPTCS.192.6

by reproducing the argument in an automated, interactive theorem prover, ACL2. The advantage of using a theorem prover is soundness and generality: by using a carefully designed and thoroughly tested theorem prover, we have high confidence in the theorems that it establishes. The critical limitation of using a theorem prover is that formulating the proofs can require large amounts of very highly skilled effort. Our solution is to integrate a SMT solver, Z3, into ACL2. This allows many parts of the proof, especially those involving large amounts of tedious algebra, to be performed automatically. While our focus is on AMS, the issues addressed here are common to those in most computing devices and other physical systems.

Our implementation uses ACL2's trusted clause processor mechanism for integrating external procedures. Our goal is to provide a flexible framework for developing proofs in a relatively new application domain. Thus, our clause processor is designed to be easily configured and modified by the user. However, too much freedom to change the behaviour of the clause processor also raises the spectre of unsoundness. We address this with a two-pronged solution. Our clause processor is available with a standard configuration, where the soundness depends mainly on the soundness of ACL2, the SMT solver, and a small amount of interface code. There is also a customizable configuration that has a separate trust-tag. This facilitates experimentation, but places the burden for soundness directly upon the user. We describe our use of the two approaches, and show how this combination provides a flexible environment for experimentation and a safe environment for "production" use.

The key contributions of this work are:

- We present our software architecture for integrating an SMT solver into ACL2 as a trusted clause processor.
- We describe the issues that arose in this integration, our solutions, and the rationale behind our design choices.
- Our emphasis is on using the arithmetic capabilities of the Z3 SMT solver. This differs from most prior work on integrating SMT solvers into theorem provers that has focused on using decision procedures for SAT, integer arithmetic, and discrete data structures.
- We show how some simple customizations of the general framework can lead to a dramatic reduction in proof effort.

The rest of this paper is organized as follows: Section 2 introduces our clause processor with three simple examples. Section 3 describes our software architecture, the issues that arise when integrating an SMT solver into ACL2, and our solutions to these issues. Section 4 describes how the SMT interface can be customized. In particular, we show how adding a simple inference engine that provides an incomplete theory of `expt` greatly simplifies our proofs for verifying properties of an AMS circuit. Sections 5 and 6 present related work and a summary of the current work respectively.

2 A Short Tour

This section presents simple theorems that can be proven using `Smtlink`. The examples here assume that the `Smtlink` book has been downloaded from:

`https://bitbucket.org/pennyan/smtlink`

and certified using `cert.pl` (see the instructions in the `README` file). Program 2.1 shows how to include the `Smtlink` book where `/dir/to/smtlink` is the directory with the `Smtlink` book. The `(tshell-ensure)` form allows `Smtlink` to invoke the SMT solver in a separate process. `Smtlink`

Program 2.1 Including the `Smtlink` book

```
(add-include-book-dir :cp "/dir/to/smtlink")
(include-book "top" :dir :cp)
(tshell-ensure)
```

Program 2.2 A theorem about a system of polynomial inequalities

```
(defthm poly-ineq-example-a
  (implies (and (rationalp x) (rationalp y)
                (<= (+ (* 4/5 x x) (* y y)) 1)
                (<= (- (* x x) (* y y)) 1))
           (<= y (- (* 3 (- x 17/8) (- x 17/8)) 3)))
  :hints (("Goal"
           :clause-processor
           (Smtlink clause nil))))

(defthm poly-ineq-example-b
  (implies (and (rationalp x) (rationalp y)
                (<= (+ (* 2/3 x x) (* y y)) 1)
                (<= (- (* x x) (* y y)) 1))
           (<= y (+ 2
                    (- (* 4/9 x))
                    (- (* x x x x))
                    (* 1/4 x x x x x x)) ))
  :hints (("Goal"
           :clause-processor
           (Smtlink clause nil))))
```

supports two configurations. The examples in this section use `Smtlink`, which uses default settings. The other, `Smtlink-custom-config`, can be configured by the user and is described in Section 4.

Program 2.2 shows two examples involving systems of polynomial inequalities: `nil` is a list of additional hints for the clause processor as no further hints are needed for these examples. Why would we want to prove such theorems? Simple, they illustrate the challenges of using ACL2 to reason about systems of polynomial inequalities as often appear in models of physical systems including AMS verification. Without the clause-processor, the proofs fail in ACL2 with the `:nonlinearp` hint enabled and with or without any of the arithmetic books (i.e. `arithmetic/top-with-meta`, `arithmetic-2/meta/top`, `arithmetic-3/top`, and or `arithmetic5/top`). Of course, a patient and savvy user could guide ACL2 through a sequence of lemmas and eventually discharge the claims. Using the SMT solver, the theorems are proven automatically.

Some theorems, while tedious to prove in ACL2, simply cannot be proven by SMT techniques alone. Consider Program 2.3. Again, when just using ACL2, the proof fails with or without a `:nonlinearp` hint or any of the arithmetic books. As formulated, `poly-of-expt-example` would appear to be unsuitable for proof with our SMT techniques because we are using Z3 as our SMT solver, and Z3 does not support reasoning about non-polynomial functions such as `expt`. Our solution is to allow the user to give hints to the clause processor. These hints allow the user to direct the clause pro-

Program 2.3 A claim with non-polynomial arithmetic

```
(defthm poly-of-expt-example
  (implies (and (rationalp x) (rationalp y) (rationalp z) (integerp m)
                (integerp n) (< 0 z) (< z 1) (< 0 m) (< m n))
           (<= (* 2 (expt z n) x y) (* (expt z m) (+ (* x x) (* y y)))))
  :hints(("Goal"
          :clause-processor
          (Smtlink clause '((:let ((expt_z_m (expt z m) rationalp)
                                   (expt_z_n (expt z n) rationalp)))
                            (:hypothesize ((< expt_z_n expt_z_m)
                                           (< 0 expt_z_m)
                                           (< 0 expt_z_n)))
                  ) ))))
```

cessor to replace all occurrences of a given expression with a new, free variable, and to express constraints that are satisfied by these variables. A complete description of these hints is presented in Section 3. To prove `poly-of-expt-example`, we use the clause-processor hint. We also include the book `arithmetic-5/top`. The two `:let` hints direct `Smtlink` to replace all occurrences of `(expt z m)` with the variable `expt_z_m`; furthermore, we are asserting that the value `(expt z m)` satisfies `rationalp`. Likewise for `expt_z_n` replacing all occurrences of `(expt z n)`. The three `:hypothesize` hints state additional constraints on the values of `expt_z_m` and `expt_z_m` for use by the SMT solver. With these substitutions and constraints, Z3 readily discharges the main claim.

For this approach to be sound, these substitions, type-assertions, and constraints must all be implied by the hypotheses of the original theorem. If the SMT solver discharges the main claim, then `Smtlink` returns each of these added assumptions and new clauses to be proven by ACL2. In other words, `Smtlink` has replaced a clause that would be difficult to prove using ACL2 alone, with a moderate number of simpler clauses that are simpler for ACL2 to establish, plus one clause (the augmented, original claim) that is proven by the SMT solver. In this case, runes from `arithmetic-5/top` enable the returned clauses to be discharged without further assistance. This also illustrates the synergies that are available by combining SMT techniques with theorem proving.

3 Software architecture of `Smtlink`

Figure 1 shows the structure of `Smtlink`. The clause processor translates ACL2 clauses into a Python representation inspired by Z3's Python API. The translation process is divided into two phases. The first phase translates from ACL2 to ACL2. This translation allows the clause-processor to accept a fairly expressive subset of the ACL2 language while the expanded clauses output by this phase use only a small set of primitive Lisp functions (See Section 3.2.2). The second phase translates the simplified (but expanded) ACL2 clauses to our Python API – this process is the main "trusted" aspect of our trusted clause processor. The SMT solver verifies a clause by showing that its negation is unsatisfiable. If this is the case, then `Smtlink` returns a list of clauses for subgoals that arose in the translation process. Essentially, `Smtlink` asks ACL2 to verify that the expanded clause implies the original, and to verify any type assertions or additional hypotheses that were provided by the user. If the SMT solver fails to show that the clause is unsatisfiable, it typically provides a counter-example that `Smtlink` then prints

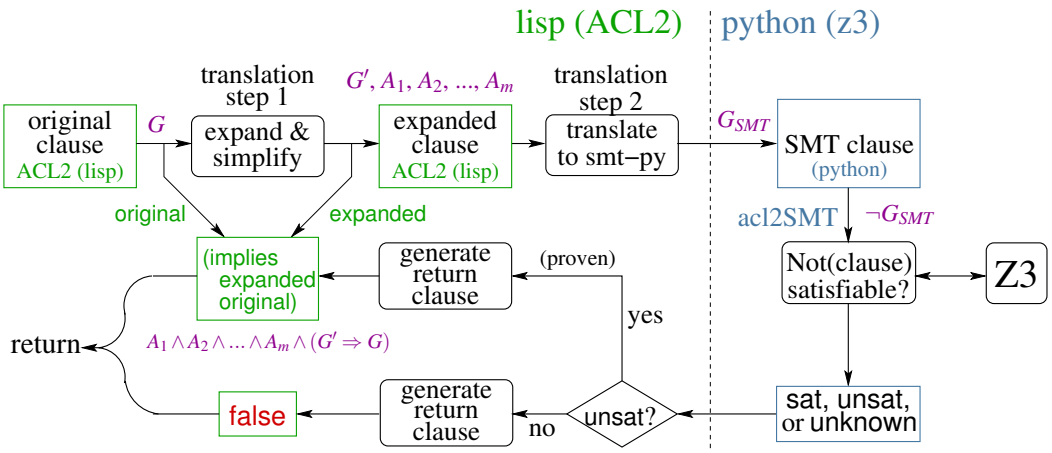


Figure 1: Top-level architecture of `Smtlink`

Program 3.1 A putative theorem without type constraints

```
1 (defthm not-really-a-theorem
2   (iff (equal x y) (zerop (- x y))) )
```

to the ACL2 comment window, although is some cases it may simply report that the satisfiability of the clause is "unknown". In these cases, `Smtlink` prints the counter-example or "unknown" status to the ACL2 comment window and aborts the proof attempt.

3.1 The first translation phase

The first phase of translation transforms clauses written in a fairly expressive subset of ACL2 into a very small subset. Most of the complexity of the translation process is in this first phase. As described in Section 3.3, `Smtlink` constructs a new clause that is proven by ACL2 to validate this translation. The key issues in the first phase are:

- ACL2 is untyped whereas SMT solvers support many-sorted logics.
- ACL2 clauses often include user-defined functions.
- The user may add type assertions and/or extra hypotheses to enable the SMT solver to discharge a claim. These must be verified by ACL2.
- The user may need to provide hints to enable ACL2 to discharge subgoals that are returned by the clause processor.

3.1.1 Types

Consider the putative theorem shown in Program 3.1. ACL2 is untyped and requires all functions to be total. Accordingly, `(- x y)` is defined for all values for `x` and `y`, including non-numeric values. As defined in ACL2, arithmetic operators such as `-` treat non-numeric values as if they were `0`. Thus, `x` =

Program 3.2 A simple theorem with type constraints

```
(defthm rational-minus-and-equal
  (implies (and (rationalp x) (rationalp y))
           (iff (equal x y) (zerop (- x y))) ))
```

`'dog` and `y = (list "hello" 2 'world)` is a counter-example to `not-really-a-theorem`. On the other hand, Z3 uses a typed logic, and each variable must have an associated sort. If we treat `x` and `y` as real-valued variables, the `z3py` equivalent to `not-really-a-theorem` is

```
>>> x, y = Reals(['x', 'y'])
>>> prove((x == y) == ((x - y) == 0))
proved
```

In other words, `not-really-a-theorem` as expressed in the untyped logic of ACL2 is not a theorem, but the "best" approximation we can make in the many-sorted logic of Z3 is a theorem. To solve these problems, `Smtlink` requires that each free variable in a theorem is constrained by an ACL2 type recognizer such as `integerp` and `rationalp`. These are then translated to corresponding SMT sorts with the design requirement that the set of values in the SMT sort must be a superset (or equal to) the set of values admitted by the type recognizer.

Although ACL2 is untyped, it is common for users to include assertions such as `(rationalp x)` that constrain the types of free-variables appearing in a theorem. Program 3.2 shows the previous, putative theorem with type recognizers added to the hypotheses. ACL2 proves `rational-minus-and-equal` without any assistance from the user. Note that `rational-minus-and-equal` holds for *all* values of `x` and `y` including values that are not rational, and values that are not even numeric, such as `x = 'dog` and `y = (list "hello" 2 'world)`. For such cases, the antecedent of the theorem is not satisfied, and the theorem holds vacuously.

Let G be the clause to be proven by `Smtlink`; G is the "goal". In the first translation phase, `Smtlink` traverses G looking for terms of the form *(typep var)* where *typep* is one of `booleanp`, `integerp`, or `rationalp`; *var* is a symbol (but not nil); and the clause holds vacuously if `(not (typep var))`. In other words, such terms are *type hypotheses*. `Smtlink` identifies type hypotheses *syntactically* by walking the tree for the expression, recognizing the constructions for `if`, `implies`, `not`, and the type-recognizers (note: the ACL2 macros "`and`" and "`or`" expand to terms written with `if`).

Let $T =$ `(list` T_1 T_2 $\ldots T_m$`)` be the list of all type-hypotheses; $\widehat{T}$ denote the conjunction of the elements of T; and and G_T be G rewritten by replacing each of the T_i's with the boolean constant `t`. We could now construct the terms $\widehat{T} \Rightarrow G_T$, $\widehat{T} \vee G$, and $((\widehat{T} \vee G) \wedge (\widehat{T} \Rightarrow G_T)) \Rightarrow G$. We could then invoke the SMT solver to determine if G_T holds for all valuations of the free variables that satisfy T. If the SMT solver can show this, then $\widehat{T} \Rightarrow G_T$ is established. Then, we could return the terms $\widehat{T} \vee G$, and $((\widehat{T} \vee G) \wedge (\widehat{T} \Rightarrow G_T)) \Rightarrow G$ to ACL2 to be proven. If these proofs are successful, then we can conclude that G is a theorem as well. `Smtlink` uses this approach; however, rather than checking each step of the first translation phase, it checks the final result. Section 3.3 describes this process.

3.1.2 Functions

The second phase of translation supports a small set of ACL2 built-in functions (see Section 3.2). `Smtlink` handles other functions by expanding their calls. In particular *(fun actual-parameters)* be-

Program 3.3 :expand hint

```
:hints(("Goal"
        :clause-processor
        (Smtlink '( ...
                   (:expand ((:functions ((fun1 type1p) (fun2 type2p) ...
                                          (funk typekp)))
                   (:expansion-level 1)))
                   ...))))
```

comes

$$((\texttt{lambda}\ (\textit{fresh-variables-for-formals})\ \textit{body-of-fun})\ \textit{actual-parameters}) \tag{1}$$

Because *body-of-fun* may have function instances that need to be expanded, `Smtlink` recursively applies this function-expansion operation to *body-of-fun* and each term in *actual-parameters*.

If the function *fun* has a recursive definition, then the expansion procedure described will not terminate. To avoid this problem, we require the user to specify a maximum expansion depth and the return type for each function. `Smtlink` replaces each call beyond the expansion limit with an unconstrained, fresh SMT variable of the specified return type. The type-hypothesis for each such variable is added to the type-hypothesis list, T, and the function call instance that this variable replaces is added to a list of function calls instances, F. As described in Section 3.3, `Smtlink` produces a clause for ACL2 to check to verify that each function call in F returns a value of the user-claimed type. Replacing the function's return value with an unconstrained variable is a simple form of generalization.

The user controls function expansion by `Smtlink` with a `:expand` hint as shown in Program 3.3. Each function is specified with its return type, and the :expansion-level parameter specifies the maximum depth to which any function will be expanded. We write G_F to denote the clause produced by expanding the function calls in G_T.

`Smtlink` also supports translating function calls in ACL2 into uninterpreted function instances. For example,

```
(:uninterpreted-functions ((expt rationalp integerp rationalp)))
```

says that the function `expt` should be treated as an unintepreted function whose first argument satisfies `rationalp`, whose second argument satisfies `integerp`, and whose return value satisfies `rationalp`. `Smtlink` records each uninterpreted function declaration in a list, U, and each call in F.

The mechanisms for function expansion and uninterpreted functions are similar. In particular, the replacement of a recursive function call with a fresh variable is a weaker version of replacing it with an uninterpreted function. On the other hand, we discovered that Z3 does not combine its theories of non-linear arithmetic and uninterpreted functions: if a formula includes an uninterpreted function, the non-linear arithmetic solver is silently disabled. Thus, in many cases, using fresh variables is preferred to using uninterpreted functions. We are examining these trade-offs in examples of real proofs and expect to formulate a more unified treatment of function expansion and uninterpreted functions in a future version of `Smtlink`.

3.1.3 Adding Hypotheses

Often, the proof of a theorem may depend on results that have already been established in ACL2's logical world. However, `Smtlink` only translates the current goal for the SMT solver. In practice, this is critical:

while it is tempting to give the SMT solver every constraint that might be relevant, this would often cause the SMT solver to require more time or memory than is available for the proof. A key feature of the integration of SMT solvers into a theorem prover is that the user can identify the *relevant* facts, and these can be included with `:hypothesize` hints as illustrated in Program 2.3. Of course, the user can include any term they like in these hints. If the SMT solver discharges the clause, then each of the `:hypothesize` hints is returned as a subgoal. If it corresponds to a previously proven theorem, then ACL2 will (usually) discharge it without any further assistance. We write H to denote the set of all hypotheses introduced by `:hypothesize` hints, $\widehat{H}$ to denote the conjunction of the elements of H, and $G_H = \widehat{H} \Rightarrow G_F = \neg\widehat{H} \vee G_F$ to denote the goal clause augmented with these hypotheses.

3.1.4 Substitutions

Proof goals may include terms that do not have a representation in the theories of the chosen SMT solver. For example, the theorem in Program 2.3 used the `expt` function that raises its first argument to an arbitrary integer power and is not representable in Z3 which only supports fixed-degree polynomials and rational functions. Rather than abandoning the advantages offered by the SMT solver, `Smtlink` allows the user to specify a replacement of offending sub-expressions by fresh variables of the appropriate types. All occurrences of the given sub-expression are replaced by the specified variable. This is another example of generalization by replacing the return value of a function with a fresh variable. It is quite common, in our experience, to combine these substitutions with `:hypothesize` hints that constrain the values of these variables. Furthermore, the type-hypothesis for each new variable is included in the type-hypotheses list, T, and the substitutions are recorded in a list S.

These substitutions are the final step of the first phase of translation. We write G' to denote the result of this first phase, and refer to it as the "expanded clause".

3.2 The second translation phase

Given an original goal, G, along with user provided hints, the first translation phase produces an "expanded goal", G'; a list of type-assertions, T; a list of functions to be treated as uninterpreted, U; a list of function call instances, F; a list of additional hypotheses, H; and a list of substitutions, S. The second translation phase uses these to produce the variable declarations for the SMT solver and the claim that the SMT solver is to discharge. If the SMT solver shows that `(not (implies` H G'`))` is unsatisfiable for valuations of the free variables that satisfy the type-hypotheses, T, and the uninterpreted function definitions, U, then `Smtlink` concludes that `(implies` H G'`)` is a theorem. Unlike the first phase, the results of the transformations performed in this second phase are not returned to ACL2 to be verified. Our design goal was to keep this part of the connection as simple as possible to avoid errors and enable code inspection by cautious users.

3.2.1 Types

For each free-variable, x_i, occurring in G (and thus in G') there should be a corresponding type-assertion, T_i that is a conjunct of T. For each type assertion, $(typep_i\ var_i)$, `Smtlink` generates a corresponding variable declaration for the SMT solver. For example,

(rationalp x)

translates to

```
x = _SMT_.isReal("x")
```

Program 3.4 The irrationality of $\sqrt{2}$

```
(defthm sqrt-of-2-is-irrational
  (implies (rationalp x) (not (equal (* x x) 2))))
```

In our implementation, the Python interface to the SMT solver is in the form of an object, _SMT_. For example, _SMT_.isReal("x") creates a real-valued, symbolic variable for the SMT solver that underlies _SMT_. If a type-assertion is omitted, then an undeclared variable will appear in the formula to be checked by the SMT solver, and the SMT solver will report an error and fail.

For soundness, if Smtlink maps the ACL2 type recognizer $typep_i$ to the SMT sort $sort_i$, then every value that satisfies $typep_i$ must be an element of $sort_i$. Note that $sort_i$ may include other values as well, this simply strengthens the claim G' and may result in a failure to prove a valid goal, but this will not cause Smtlink to prove an invalid goal. Smtlink maps the ACL2 type recognizers booleanp to SMT booleans; the type correspondence is strict. The type recognizers integerp and rationalp are both mapped to SMT reals. We did this because most SMT solvers (e.g. Z3) provide decision procedures for real numbers, whereas ACL2 provides rationals. As noted above, this strengthens the claim. For example, the theorem shown in Program 3.4 can be proven in ACL2 [10], we cannot discharge it using Smtlink. It will report the counter-example for x equal to the square-root of two, described as an algebraic number. Smtlink can also be used with ACL2r, in which case the mismatch between rationals and reals can be avoided entirely.

Our choice to broaden integerp to SMT rationals (instead of SMT integers) was pragmatic. Our initial implementation uses the Z3 SMT solver, and we make extensive use of its non-linear arithmetic solver. Z3 disables the non-linear solver when a formula includes integer-valued variables. By mapping ACL2 integers to SMT reals, Smtlink strengthens the theorem. We expect to add mechanisms to allow the user to control whether ACL2 integers map to SMT integers or reals in a future version of Smtlink.

3.2.2 Functions

The nine functions supported are binary-+, unary--, binary-*, unary-/, equal, <, if, not, and lambda along with the constants t, nil, and arbitrary integer constants. As in ACL2, integers in Python can be arbitrarily large; thus, Smtlink translates them directly. Smtlink translates ACL2 lambda expressions into Python lambda expressions. The other eight functions are translated directly to their counterpart methods of the _SMT_ object. For example, the ACL2 function binary-+ is mapped to _SMT_.plus. Smtlink generates declarations for all uninterpreted functions, again using the _SMT_ interface.

If G' includes any functions that are not in the list of eight above or in U, then Smtlink will not prove G but instead will fail with an error message. In particular, unexpanded occurrences of user-defined functions will create an error. Furthermore, any type-recognizer such as rationalp in G' will create an error – Smtlink requires that all type-recognizer terms occur in contexts that it can recognize as type-hypotheses; others generate errors. Likewise, G cannot include quantification operators such as exists or forall. This ensures that all variables appearing in G' are free which is essential for our approach of using SMT sorts that are super-sets of their ACL2 equivalents. For example, one cannot state a theorem that 2 has a rational square root and "prove" it using Smtlink to find a real-valued x such that x*x = 2.

In the SMT world, each operation (such as +) is defined for specific sorts for its arguments and

defined to to produce a (symbolic) value of a specific sort for its result. Some of these operations (such as +) are overloaded to operate on multiple types. If an operator is applied to arguments for which it is not defined, then the SMT solver fails, and `Smtlink` fails to prove the goal. For example, if the original goal, G, (and thus G') includes a term of the form `(+ x b)` where `x` is real and `b` is boolean, then the SMT solver will fail even though the operation is defined in ACL2. This interpretation of ACL2 operators is conservative: `Smtlink` will not discharge an invalid theorem due to the type restrictions of operators in the SMT world.

`Smtlink` translates `(/ m)` in ACL2 to `_SMT_.reciprocal(m)`, where the SMT function divides the constant 1 by m. If $m \neq 0$, the ACL2 and SMT operations are identical. If $m = 0$, then the SMT version produces an unconstrained integer (if m is an integer) or real (if m is real). The ACL2 operator is defined to return 0. Because the SMT version allows the ACL2 semantics, the SMT version is more general. Thus, `Smtlink` proves a more general claim, and a proof of G' implies a proof of G. This relies on our restriction that G cannot include quantification operators.

3.2.3 Hypotheses and Substitutions

These are handled entirely in the transformation of the original goal, G, to the expanded goal, G', in the first translation phase and do not impact the second phase.

3.3 Ensuring soundness

Our design goal with `Smtlink` has been to trust ACL2, the chosen SMT solver (Z3, in our current implementation), and as little other code as practical. At the same time, our intended use for `Smtlink` is for the verification of AMS circuits and other cyber-physical systems. Because we are developing verification techniques as we go, we want `Smtlink` to provide a flexible framework for prototyping new ideas. Our solution is to put most of the functionality and complexity of `Smtlink` into the first translation phase. If the SMT solver discharges the translated clause, then `Smtlink` generates a set of return clauses to check the correctness of this translation. The second phase is trusted; this code is both small and simple.

Our basic approach is simple: let A denote the additional assumptions that were added to the goal by type assertions for variables and function return values, hypotheses, and substitutions. Let G_{SMT} denote the clause that is tested by the SMT solver. If the SMT solver proves G_{SMT}, then `Smtlink` returns the clauses

$$\begin{array}{lcl} Q_1 & = & (G' \wedge A) \Rightarrow G \\ Q_2 & = & A \vee G \end{array} \tag{2}$$

for proof by ACL2. We are trusting the translation of G' to G_{SMT} and the SMT solver itself, Modulo that trust, the truth of G_{SMT} implies the truth of G'; in which case Q_1 is equivalent to $A \Rightarrow G$. Accordingly, when ACL2 proves Q_1 and Q_2, G is established as a theorem.

We make two observations before describing how each step of the translation process contributes to A. First, the correctness of this argument does not depend on the choice of A. Of course, deriving the intended A is important to ensure that Q_1 and Q_2 can actually be proven. Second, A is the conjunction of the various assumptions that were added by `Smtlink`. `Smtlink` expresses Q_2 as a separate subgoal for each conjunct of A.

3.3.1 Types

Each type-hypothesis identified by `Smtlink` is included in A. Let $T_i = (typep_i\ var_i)$ be such a type-hypothesis. When proving Q_2, ACL2 verifies $T_i \vee G$ which means that for all values of var_i that do not satisfy $typep_i$, G trivially holds. By the trust that `Smtlink` declares var_i to be of an SMT sort that includes all values that satisfy $typep_i$, the translation is valid.

3.3.2 Functions

When a function call is expanded in the first translation phase, the equivalence is checked by ACL2 when it verifies Q_1. We are trusting the translation of ACL2 lambda-expressions to their Python equivalents in phase 2. When a function call is replaced by a variable, ACL2 must check that the user-claimed type for the return value of the function is valid. This is done by generating a clause for each function call in F. Let f be such a function call (i.e. an ACL2 term), and let $type_f$ be the user-claimed type for the return value of f. `Smtlink` includes a conjunct of the form

$$(\texttt{or}\ (type_f\ f)\ G) \tag{3}$$

in Q_2. A technical detail is that f may include variables that are bound by lambda expression arising from other function expansions; such variables are free in the clause depicted in Equation 3 as generated by `Smtlink`. This means that these variables are less constrained in the check performed by ACL2 than they are in G' or G_{SMT}. Because ACL2 has proven the more general case, we can safely conclude the more restricted version as well.

3.3.3 Added Hypotheses

Each hypothesis, H_i, added by the user, is included in A. The clause $(H_i \vee G)$ is verified by ACL2; therefore, it is safe to add H_i as a hypothesis for G' (and thus for G_{SMT}).

3.3.4 Substitutions

`Smtlink` records the user-defined substitutions in the list S. When generating Q_1 and Q_2, `Smtlink` uses lambda expressions to bind the variables declared in substitution hints to their corresponding expressions – this is similar to the way that function expansions are handled. Furthermore, the user-claimed types of these expressions are included in T, and `Smtlink` generates clauses for ACL2 to check these claims in the same manner as checking the types of values returned by function calls.

3.3.5 The Python Interface

`Smtlink` relies on software packages that are outside the ACL2 world, namely the Python interpreter and an SMT solver (Z3 for the purposes of this paper). This creates the potential unsoundness that these external components can be modified without detection. Our implementation of `Smtlink` takes several measures to prevent the most likely causes of unsoundness. First, `Smtlink` has a default configuration that is encoded in `config.lisp`. There is a script for creating `config.lisp`; once run, the configuration includes full path names to the Python interpreter and sets the path variable for searching for Python classes. Likewise, the Python code to define the class for the interface object, `_SMT_` described in Section 3.2 is provided as the string returned by the function ACL22SMT. The file ACL22SMT.lisp is generated from a Python source file that is specific for the intended SMT solver. The consequence of

this approach is that the paths to the Python interpreter and the SMT solver (and therefore the choice of the SMT solver), along with the Python class definition for the interface between `Smtlink` and the SMT solver are all baked into the certified ACL2 code for `Smtlink`. We believe that this should make `Smtlink` quite robust to unintentional changes of the computing environment. Of course, a nefarious user could replace the executable image for the Python interpreter, or the dynamic library for the SMT solver, but these are in "system" directories (under `/usr/bin` in our installation) rather than user directories; so such changes are unlikely to be accidental. Such changes *are* likely to occur as a consequence of regular software updates. We are considering adding checksum information to our `config.lisp` to ensure that such changes are detected and reported. We would like to devise an SMT-solver independent way of recording such checksums.

3.3.6 Remarks

In the current implementation of `Smtlink`, the construction of the goals Q_1 and Q_2 is done within the trusted code of the clause processor. Although the arguments for the correctness of these constructions are straightforward, the fact that this is unverified code does present a risk of errors. As we have learned more about ACL2, we now see that an alternative would be to restructure `Smtlink` to provide a function that returns G' and the lists T, F, U, H, and S described above. From these, a local theorem, that G' holds would be proven using a trusted clause processor corresponding to phase 2 of the current `Smtlink`. Additional local theorems would be proven by ACL2 to prove Q_1 and each clause of Q_2 from Equation 2. Then, the main theorem, G would be proven by ACL2 using these local theorems. This should be a relatively straightforward restructuring `Smtlink` that would isolate the small amount of trusted code. We plan to do this in the near future.

Even greater confidence could be achieved by adopting the "skeptical" approach advocated by Harrison [11], for example by using proof reconstruction [6, 9, 17, 2, 18] or proof certificates [5]. We see such efforts as complementary to the approach that we have taken with `Smtlink`. We are using `Smtlink` to develop proof methods for domains where formal methods have had little prior use. As described in Section 4, the relatively lightweight interfaces in `Smtlink` facilitate such experimentation. We gain this flexibility at the risk that an error in critical parts of our code (or in the SMT solver itself) could lead to a "proof" of a non-theorem. We believe that this risk is small compared with other risks that are inherent in the verification of physical artifacts: most notably, "Does the model of the physical system actually capture all possible behaviours?" Being able to prototype and develop proofs quickly lets us explore the consequences of the models more thoroughly than would be possible with a less flexible approach. Thus, we regard the slight risk of an error as being justified by the opportunity to verify designs that are otherwise outside the reach of formal tools. We see this as complementary to work on proof reconstruction and proof certificates. If we demonstrate the kinds of proofs that are useful in practice, that should illuminate where proof reconstruction and certificates would offer the greatest increase in confidence in critical designs.

4 Customizing `Smtlink`

The design choices described in Section 3.3.5 protect the user from unintentional changes to the external components of `Smtlink`. What if such changes are desired? To facilitate such experimentation, we provide a second version of `Smtlink`, `Smtlink-custom-config`, where the user can easily change the configuration of external components. Using `Smtlink-custom-config` requires a different trust-

Table 1: Rules for `expt`

1.	`(expt x 0)` $\rightarrow$ `1`	
2.	`(expt 0 n)` $\rightarrow$ `0`,	if $n > 0$
3.	`(expt x (+ n1 n2))` $\rightarrow$ `(* (expt x n1) (expt x n2))`	
4.	`(expt x (* c n))` $\rightarrow$ `(* (expt x n) (expt x n) ... (expt x n))`	
5.	`(< (expt x m) (expt x n))`, if $1 < x$ and $m < n$	
6.	`...`	

Notes: All rules have a precondition of that either the base is non-zero or the exponent is positive; furthermore, new instances of `expt` are only generated if they can be shown to satisfy the same condition. For rule 4, the right-hand side of $\rightarrow$ is the multiplication of `c` copies of `(expt x n)`. Rule 4 is only applied if `c` is small and positive.

tag than that for the standard configuration, `Smtlink`. Thus, it is easy to track theorems whose proofs descend from a custom configuration of the clause processor. The remainder of this section describes one such custom configuration to illustrate how these features facilitate experimentation.

Our largest use of `Smtlink` to date has been the proof of global convergence for a digital phase-locked loop (The code can be found at [19] and see [20] for more details.). The original proof was a 13 page long latex document, with lots of tedius algebra. Using the standard configuration of `Smtlink`, we completed the same proof using ACL2. The proof is about 1700 lines of ACL2 code. While `Smtlink` made the proof possible, it didn't make it as easy as we had hoped. A key complication is that the phase-locked loop (PLL) model uses recurrence functions whose solutions make extensive use of ACL2's `expt` function. As described earlier, `Smtlink` can handle these, but each occurrence requires `:let` and `:hypothesize` hints. Furthermore, function expansion renames variables; so, the proofs involved many lemmas whose sole purpose was to explicitly expand functions and rewrite terms so as to make the calls to `expt` visible in the theorem statement and thus amenable to these hints.

Our solution was to define a new Python class for the `_SMT_` interface object. This class is called `RewriteExpt`, and it extends the default `ACL2_to_Z3` that was complied into `ACL22SMT.lisp` as described above. To use this extension, `expt` is declared to be an uninterpreted function. `RewriteExpt` overrides the `_SMT_.prove` method to add a pre-processing step finds instances of `expt` in the claim. For each instance, the code checks to see if the hypotheses of the theorem imply the guard for `expt`: the base must be non-zero, or the exponent must be non-negative. If the guard can't be proven, an error is reported and the proof fails. Otherwise, `RewriteExpt` applies a small number of simple proof rules about `expt`. If the antecedent of one of these rules is satisfied, then the consequent is added as a new hypothesis. Table 1 shows some examples of these rules.

Preliminary experiments with this customized clause processor have been very promising. For example, one theorem in the PLL proof that required 19 supporting lemmas for a total of 334 lines of ACL2 code was replaced by a single theorem stated in 13 lines of ACL2 code. The proofs with the customized clause processor are much shorter, much simpler, and much easier to understand.

We are in the process of writing a new proof for the PLL based on the customized clause processor. We see many directions that we could pursue to extend this approach after revising the PLL proof. First, the customized clause processor uses a set of proof-rules that are hard-coded into `RewriteExpt.py`. These correspond to runes for existing ACL2 theorems about `expt`. We expect that we could forward such runes from ACL2 to the SMT interface and write a simple, generic inference engine in Python.

The advantage of performing the inference with the SMT solver is that it can discharge pre-conditions for runes that ACL2 does not resolve with its waterfall. On the other hand, the ACL2 framework is much more general than what can be described in the theories of an SMT solver; so we see the two as complementary. We also note that once our inference engine has discovered a useful hypothesis, it also has the justification. Thus, we could return these to ACL2 and use them to generate the `:let` and `:hypothesize` needed to discharge the goal with the standard configuration of `Smtlink`. If this approach were implemented, then our customized processor would be an elaborate computed hint, but the goal would be discharged with `Smtlink`, and no additional trust would be required.

5 Related work

There has been extensive work in the past decade on integrating SAT and SMT solvers into theorem provers. Srinivasan [22] integrated the Yices [8] SMT solver into ACL2 for verifying bit-level pipelined machines. They also use the mechanism of a trusted clause processor with a translation process quite similar to ours. They appear to have mostly used the bit-vector arithmetic and SAT solving capabilities of Yices. While they also produce an expanded formula that is then translated to SMT-LIB [4], they don't describe using ACL2 to check this translation as we have done. Prior to that, in [16], they integrated a decision procedure called UCLID [14] into ACL2 to solve a similar problem.

Works on integrating SMT solvers or techniques into other theorem provers include [17, 9, 5, 2, 18, 6, 7]. Many of these papers have followed Harrison and Théry's "skeptical" approach [11] and focused on methods for verifying SMT results within the theorem prover using proof reconstruction, certificates, and similar methods. Several of the papers showed how their methods could be used for the verification of concurrent algorithms such as clock synchronization [9], and the Bakery and Memoir algorithms [18]. While [9] used the CVC-Lite [3] SMT solver to verify properties of simple quadratic inequalites, the use of SMT in theorem provers has generally made light use of the arithmetic capability of such solvers. In fact [6] (Isabelle/Sledgehammer with Z3) reported better results for SMT for several sets of benchmarks when the arithmetic theory solvers were disabled!

The work that resembles our approach is [7]; they present a translation of Event-B sequents from Rodin [1] to the SMT-LIB format [4]. Like our work, [7] verifies a claim by using a SMT solver to show that its negation is unsatisfiable. They address issues of types and functions. They perform extensive rewriting using Event-B sequents, and then have simple translations of the rewritten form into SMT-LIB. While noting that proof reconstruction is possible in principle, they do not appear to implement such measures. The main focus of [7] is supporting the set-theoretic constructs of Event-B. In contrast, our work shows how the procedures for non-linear arithmetic of a modern SMT solver can be used when reasoning about analog and mixed-signal circuits.

Our work demonstrates the value of theorem proving combined with SMT solvers for verifying properties that are characterized by functions on real numbers and vector fields. Accordingly, the linear and non-linear arithmetic theory solvers have a central role. As our concern is bringing these techniques to new problem domains, we deliberately take a pragmatic approach to integration and trust both the theorem prover and the SMT solver.

Prior work on using theorem proving methods to reason about dynamical systems includes [13] which uses the Isabelle theorem prover to verify bounds on solutions to simple ODEs from a single initial condition. Harutunian [12] presented a very general framework for reasoning about hybrid systems using ACL2 and demonstrated the approach with some very simple examples. Here we demonstrate that by discharging arithmetic proof obligations using a SMT solver, it is practical to reason about much realistic

designs.

6 Conclusion and future work

This paper presented `Smtlink`, a clause-processor that we have used to integrate the Z3 SMT solver into ACL2. Reasoning about systems of polynomial and rational function equalities and inequalities can be greatly simplified by using Z3's non-linear arithmetic capabilities. ACL2 complements Z3 by providing a versatile induction capability along with a mature environment for proof development and structuring. `Smtlink` offers two configurations: the default, standard configuration where the interface code and the pathways to the external tools (Python and Z3) are fixed when book is certified; and a customizable interface that allows the user to experiment with extending these capabilities.

Section 3 described our software architecture, issues that arose when integrating an SMT solver into ACL2, and our solutions to these issues. A key aspect of the design is a two-phase translation process for converting ACL2 clauses into formulas that can be discharged by the SMT solver. The first phase translates a fairly expressive subset of ACL2 into a simple subset consisting of nine built-in functions. This first phase includes methods for handling types, function expansion, uninterpreted functions, and sub-expression replacement; all of these can be understood as various versions of generalizing the original clause to produce a stronger clause that is suitable for discharging with an SMT solver. Most of the complexity of the translation process is in the first phase. Because ACL2 verifies that the clause produced by this first phase implies the original, this first phase greatly improves the usability of the clause processor while raising minimal concerns about soundness. The second phase transliterates the nine remaining functions to equivalents in a Python API – this is the code that is most critical for sondness.

Section 4 showed how the customizable interface can be used to automate tedious aspects of a moderately large (1700 line) proof that we performed with the original version of the clause processor. By adding a few simple rules for transforming expressions involving the ACL2 `expt` functions into the SMT interface, we showed that we could dramatically reduce the length and complexity of some of the proofs. We believe that this demonstrates the value of `Smtlink` as an experimental platform. Once a proposed functionality is shown to have sufficient value, then a more rigorous version could be implemented. The fast prototyping that is enabled by `Smtlink` can help guide this process by avoiding investing large amounts of effort on some approach that ultimately provides small improvements to the proof development process.

Prior work on integrating SMT solvers into theorem provers has focused on using the non-numerical decision procedures of an SMT solver. Our work focuses on the value of bringing an SMT solver into a theorem prover for reasoning about systems where a digital controller interacts with a continuous, analog, physical system. The analysis of such systems often involves long, tedious, and error-prone derivations that primarily use linear algebra and polynomials. These are domains where SMT solvers combined with induction and proof structuring have great promise.

6.1 Future work

`Smtlink` returns clauses to ACL2 to check the translation of the original goal to a small subset of ACL2. As noted in Section 3.3.6, a moderate restructuring of this code could allow most of this work to be done within ACL2 and reduce the amount of code that must be trusted in `Smtlink`. We believe that this could be done with minimal impact on the flexibility of `Smtlink` for experimenting with SMT solvers and their applications.

We have used ACL2 with `Smtlink` to prove the most challenging part of a global convergence argument for a digital Phase-Locked Loop (PLL) using `Smtlink`. Global convergence is a response property, and we can show that the PLL makes progress through four distinct phases. We used ACL2 with `Smtlink` to verify the phase for which a hand-written proof was the most complicated. We would like to write proofs in ACL2 for the other three phases and use ACL2 to prove that those results are sufficient to prove correct convergence from any initial condition. This will involve constructing Skolem functions to compose the individual pieces of the proof and should demonstrate the strength of using ACL2 to prove properties that cannot be expressed in the logic of the SMT solver.

We would like to add a bounded model checking capability to SMT link. For example, in the PLL proof, there is a tedious proof at the boundary of two of the phases. Z3 provides an "easy" proof by showing that within eight steps of the recurrence, the transition between phases is complete and correct. We would lke to integrate this capability into `Smtlink` and thus into ACL2.

The current implementation of `Smtlink` provides very restricted support for recursive functions. This is because most recursive functions are non-numerical and/or use "fixing" functions or recognizers to ensure termination when called with bogus arguments. While this has not been problematic for our PLL proof, we would like to generalize the handling of types by `Smtlink` to allow a wider range of applications.

Many of the type-checking steps performed by `Smtlink` reconstruct facts that are already present in ACL2s `type-alist`. We would like to see if this information could be used by `Smtlink` and thus spare the user of many of the type declaration that `Smtlink` now requires.

Presently, `Smtlink` prints counter-examples from the SMT solver to the ACL2 comment window. We would like to make them available to the user within the ACL2 environment. This could be similar to the `env$` argument used in `Satlink`. New issues arise in the SMT case because SMT formulas don't have a single, syntactical form like CNF for SAT. Furthermore, if the counter-example included irrational numbers, then it cannot be represented in ACL2 – although this should be addressed in ACL2(r).

Acknowledgments

We would like to thank the many members who have patiently answered our many questions while developing `Smtlink`, especially Matt Kaufmann and David Rager. We are also thankful to the anonymous reviewers for the insightful and inspiring feedback.

References

[1] J.-R. Abrial, M. Butler, S. Hallerstede & L. Voisin (2006): *An Open Extensible Tool Environment for Event-b*. In: *8th Int'l. Conf. Formal Methods and Software Engineering*, Springer, pp. 588–605, doi:10.1007/11901433_32.

[2] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry & B. Werner (2011): *A Modular Integration of SAT/SMT Solvers to Coq Through Proof Witnesses*. In: *1st Int'l. Conf. Certified Programs and Proofs*, Springer, pp. 135–150, doi:10.1007/978-3-642-25379-9_12.

[3] C. Barrett & S. Berezin (2004): *CVC Lite: A New Implementation of the Cooperating Validity Checker*. In: *Computer Aided Verification*, *LNCS* 3114, Springer, pp. 515–518, doi:10.1007/978-3-540-27813-9_49.

[4] C. Barrett, A. Stump & C. Tinelli (2010): *The SMT-LIB Standard: Version 2.0*. `http://www.cs.nyu.edu/~barrett/pubs/BST10.pdf`. [Online; accessed 17-August-2015].

[5] F. Besson (2007): *Fast Reflexive Arithmetic Tactics the Linear Case and Beyond*. In: *2006 Int'l. Conf. Types for Proofs and Programs*, Springer, pp. 48–62, doi:10.1007/978-3-540-74464-1_4.

[6] J.C. Blanchette, S. Böhme & L.C. Paulson (2013): *Extending Sledgehammer with SMT Solvers*. *J. Automated Reasoning* 51(1), pp. 109–128, doi:10.1007/s10817-013-9278-5.

[7] D. Déharbe, P. Fontaine, Y. Guyot & L. Voisin (2014): *Integrating SMT Solvers in Rodin*. *Sci. Comput. Program.* 94(P2), pp. 130–143, doi:10.1016/j.scico.2014.04.012.

[8] B. Dutertre (2014): *Yices2.2*. In: *Computer Aided Verification*, *LNCS* 8559, Springer, pp. 737–744, doi:10.1007/978-3-319-08867-9_49.

[9] P. Fontaine, J.-Y. Marion, S. Merz, L.P. Nieto & A. Tiu (2006): *Expressiveness + Automation + Soundness: Towards Combining SMT Solvers and Interactive Proof Assistants*. In: *12th Int'l. Conf. Tools and Algorithms for the Construction and Analysis of Systems*, Springer, pp. 167–181, doi:10.1007/11691372_11.

[10] R.A. Gamboa (1997): *Square Roots in ACL2: A Study in Sonata Form*. Technical Report, University of Texas at Austin. Available at `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.57.4803`.

[11] J. Harrison & L. Théry (1998): *A Skeptic's Approach to Combining HOL and Maple*. *J. Automated Reasoning* 21(3), pp. 279–294, doi:10.1023/A:1006023127567.

[12] Shant Harutunian (2007): *Formal Verification of Computer Controlled Systems*. Ph.D. thesis, University of Texas, Austin. Available at `https://www.lib.utexas.edu/etd/d/2007/harutunians68792/harutunians68792.pdf`.

[13] F. Immler (2014): *Formally Verified Computation of Enclosures of Solutions of Ordinary Differential Equations*. In: *NASA Formal Methods*, *LNCS* 8430, Springer, pp. 113–127, doi:10.1007/978-3-319-06200-6_9.

[14] S.K. Lahiri & S.A. Seshia (2004): *The UCLID Decision Procedure*. In: *Computer Aided Verification*, *LNCS* 3114, Springer, pp. 475–478, doi:10.1007/978-3-540-27813-9_40.

[15] K. Leino & Rustan M. (2012): *Automating Induction with an SMT Solver*. In: *Verification, Model Checking, and Abstract Interpretation*, *LNCS* 7148, Springer, pp. 315–331, doi:10.1007/978-3-642-27940-9_21.

[16] P. Manolios & S.K. Srinivasan (2006): *A Framework for Verifying Bit-Level Pipelined Machines Based on Automated Deduction and Decision Procedures*. *J. of Automated Reasoning* 37(1-2), pp. 93–116, doi:10.1007/s10817-006-9035-0.

[17] S. Mclaughlin, Cl. Barrett & Y. Ge (2006): *Cooperating theorem provers: A case study combining HOL-Light and CVC Lite*. In: *In Proc. 3rd Workshop on Pragmatics of Decision Procedures in Automated Reasoning*, *ENTCS* 144(2), Elsevier, pp. 43–51, doi:10.1016/j.entcs.2005.12.005.

[18] S. Merz & H. Vanzetto (2012): *Automatic Verification of TLA$^+$; Proof Obligations with SMT Solvers*. In: *18th Int'l. Conf. Logic for Programming, Artificial Intelligence, and Reasoning*, Springer, pp. 289–303, doi:10.1007/978-3-642-28717-6_23.

[19] Y. Peng (2015): *Global convergence proof for a digital Phase-Locked Loop*. `https://bitbucket.org/pennyan/smtlink/src/7fdd38280be9e492a96947019f9b0c8cf10b3d91/examples/DPLL/DPLL_proof.lisp?at=master`. [Online; accessed 17-August-2015].

[20] Y. Peng & M. Greenstreet (2015): *Integrating SMT with Theorem Proving for Analog/Mixed-Signal Circuit Verification*. In: *NASA Formal Methods*, *LNCS* 9058, Springer, pp. 310–326, doi:10.1007/978-3-319-17524-9_22.

[21] E. Reeber & W.A. Hunt Jr. (2006): *A SAT-Based Decision Procedure for the Subclass of Unrollable List Formulas in ACL2 (SULFA)*. In: *Automated Reasoning*, *LNCS* 4130, Springer, pp. 453–467, doi:10.1007/11814771_38.

[22] S.K. Srinivasan (2007): *Efficient Verification of Bit-level Pipelined Machines Using Refinement*. Ph.D. thesis, Georgia Institute of Technology.

[23] S. Swords & J. Davis (2011): *Bit-Blasting ACL2 Theorems*. In: *10th Int'l. Workshop on the ACL2 Theorem Prover and its Applications*, pp. 84–102, doi:10.4204/EPTCS.70.7.

[24] M. Weiser (1999): *The Computer for the 21st Century*. *SIGMOBILE Mob. Comput. Commun. Rev.* 3(3), pp. 3–11, doi:10.1145/329124.329126.

Reasoning About LLVM Code Using Codewalker

David S. Hardin

Advanced Technology Center
Rockwell Collins
Cedar Rapids, IA, USA
david.hardin@rockwellcollins.com

This paper reports on initial experiments using J Moore's Codewalker to reason about programs compiled to the Low-Level Virtual Machine (LLVM) intermediate form. Previously, we reported on a translator from LLVM to the applicative subset of Common Lisp accepted by the ACL2 theorem prover, producing executable ACL2 formal models, and allowing us to both prove theorems about the translated models as well as validate those models by testing. That translator provided many of the benefits of a pure decompilation into logic approach, but had the disadvantage of not being verified. The availability of Codewalker as of ACL2 7.0 has provided an opportunity to revisit this idea, and employ a more trustworthy decompilation into logic tool. Thus, we have employed the Codewalker method to create an interpreter for a subset of the LLVM instruction set, and have used Codewalker to analyze some simple array-based C programs compiled to LLVM form. We discuss advantages and limitations of the Codewalker-based method compared to the previous method, and provide some challenge problems for future Codewalker development.

1 Introduction

In previous work [9] [11], we built a translator from Low-Level Virtual Machine (LLVM) intermediate form [16] to the applicative subset of Common Lisp [15] accepted by the ACL2 theorem prover [12], and performed verification on the translated form using ACL2's automated reasoning capabilities.

LLVM is the intermediate form for many common compilers, including the `clang` compiler used by Apple OS X and iOS developers. LLVM supports a number of language frontends, and LLVM code generation targets exist for a wide variety of machines, including both CPUs and GPUs. LLVM is a register-based intermediate language in Static Single Assignment (SSA) form [4]. As such, LLVM supports any number of registers, each of which is only assigned once, statically (dynamically, of course, a given register can be assigned any number of times). Andrew Appel has observed that "SSA form is a kind of functional programming" [1]; this observation, in turn, inspired us to build a translator from LLVM to the applicative subset of Common Lisp accepted by the ACL2 theorem prover. Our translator, written in OCaml [5], produced an executable ACL2 specification that was able to support proof-based verification, as well as validation via testing.

The above approach was satisfactory for the technology that we had at hand for use with ACL2 in 2013, but had the obvious weakness of relying on a fair amount of unverified code. The situation changed in late 2014, when J Moore released the initial version of Codewalker, an instruction-set-neutral decompilation-into-logic system, with ACL2 7.0 [18]. Thus, an experiment began in early 2015 to determine whether Codewalker could be used to produce a similar proof environment for LLVM code.

M. Kaufmann and D. Rager (Eds.): ACL2 Workshop 2015 (ACL2 2015).
EPTCS 192, 2015, pp. 79–92, doi:10.4204/EPTCS.192.7

```
unsigned long occurrences(unsigned long val, unsigned int n,
                          unsigned long *array) {
  unsigned long num_occur = 0;
  unsigned int j = 0;
  for (j = 0; j < n; j++) {
    if (array[j] == val) num_occur++;
  }
  return num_occur;
}
```

Figure 1: Example C code to count occurrences of an input value in an array.

2 An Example

As an example, consider the C source code of Figure 1. This function counts the number of occurrences of a given value in the first n elements of an array. (NB: By default the `clang` compiler treats all `int` values as 32 bits wide, and all `long` values as 64 bits wide.)

This is an admittedly simple example, but it allows us to narrate a complete analysis within the confines of this paper, and should be within Codewalker's capabilities to analyze. We have also performed similar analyses for other small C programs, namely tail-recursive factorial, as well as a program to compute the sum of array elements.

LLVM code for this function is produced by invoking `clang` as follows: `clang -O1 -S -emit-llvm occurrences.c`. The generated LLVM code for clang version 6.1.0 (which supports LLVM 3.6.0) is excerpted in Figure 2; this is essentially the same code as reported in [9].

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

2.1 Translation to ACL2 Syntax

In previous work, we automatically translated the above LLVM program into an ACL2 functional program. In the current work, we merely translate the LLVM assembly code syntax into a form that is easier for ACL2 to process. The translated form for the LLVM code of Figure 2 is depicted in Figure 3.

The instruction format is straightforward: if the LLVM instruction is `a = ins b c`, then the ACL2 syntax is `(INS A B C)`. Thus, `(ADD x y z)` stores the sum of the contents of registers (locals) y and z in register x; and `(BR E F G)` branches to the instruction word at the current program counter + offset F if register E is nonzero, and to the instruction word at the current program counter + offset G otherwise. A few new instructions have been added to aid in phi processing: `(CONST X)` pushes a constant value X on a LIFO stack; `(PUSH Y)` pushes the contents of register Y onto the stack; and `(POPTO Z)` pops the top of stack value into register Z. We also define a `(HALT)` instruction so we don't have to worry about defining a return linkage (this is future work).

Each instruction occupies one instruction word (of indeterminate size), and each register holds an

```
define i64 @occurrences(i64 %val, i32 %n, i64* %array) {
  %1 = icmp eq i32 %n, 0
  br i1 %1, label %._crit_edge, label %.lr.ph

.lr.ph:
  %indvars.iv = phi i64 [ %indvars.iv.next, %.lr.ph ], [ 0, %0 ]
  %num_occur.01 = phi i64 [ %.num_occur.0, %.lr.ph ], [ 0, %0 ]
  %2 = getelementptr inbounds i64* %array, i64 %indvars.iv
  %3 = load i64* %2, align 8, !tbaa !1
  %4 = icmp eq i64 %3, %val
  %5 = zext i1 %4 to i64
  %.num_occur.0 = add i64 %5, %num_occur.01
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %lftr.wideiv = trunc i64 %indvars.iv.next to i32
  %exitcond = icmp eq i32 %lftr.wideiv, %n
  br i1 %exitcond, label %._crit_edge, label %.lr.ph

._crit_edge:
  %num_occur.0.lcssa = phi i64 [ 0, %0 ], [ %.num_occur.0, %.lr.ph ]
  ret i64 %num_occur.0.lcssa
}
```

Figure 2: LLVM code for the occurrences example.

```
    ;;    reg[2] contains val
    ;;    reg[1] contains n
    ;;    reg[0] contains array base address


    (CONST 0)           ; 0
    (POPTO 3)           ; 1   reg[3] <- 0
    (EQ 4 1 3)          ; 2   n == 0?

    (CONST 0)           ; 3
    (POPTO 5)           ; 4   phi(j), j <- 0
    (CONST 0)           ; 5
    (POPTO 6)           ; 6   phi(num_occur), num_occur <- 0

    (BR 4 14 1)         ; 7   branch to ._crit_edge if n == 0

;; .lr.ph:
    (GETELPTR 7 0 5)    ; 8   reg[7] <- mem address of arr[index]
    (LOAD 8 7)          ; 9   reg[8] <- mem[reg[7]] = arr[index]
    (EQ 9 8 2)          ; 10  reg[8] == val?
    (ADD 10 6 9)        ; 11  num_occur conditional increment
    (CONST 1)           ; 12
    (POPTO 11)          ; 13
    (ADD 12 5 11)       ; 14  reg[12] <- j+1
    (EQ 13 12 1)        ; 15  j+1 == n?

    (PUSH 12)           ; 16
    (POPTO 5)           ; 17  phi(j), j <- j+1
    (PUSH 10)           ; 18
    (POPTO 6)           ; 19  phi(num_occur)

    (BR 13 1 -12)       ; 20  loop back to .lr.ph if j+1 < n

;;._crit_edge:
    (PUSH 6)            ; 21  push num_occur on stack
    (HALT)              ; 22
```

Figure 3: ACL2 representation of the LLVM code for the occurrences example.

unbounded integer. This represents a slight loss of fidelity relative to the previous work, but we thought it unwise to tackle issues related to both Codewalker and modular arithmetic at the same time.

3 LL2: An LLVM Subset Interpreter

Before being able to utilize Codewalker, we must first define an operational semantics, or interpreter, for the target instruction set. The Codewalker sources provide one such example interpreter, for the M1 subset of the Java Virtual Machine (JVM) [14]. We used this ACL2 code as the basis for our LLVM subset interpreter, called LL2. As is typical with such an interpreter written in ACL2, a machine state data structure is declared, and passed as a parameter to all functions that read and/or write elements of the state. If a given function updates the state, the modified state must be returned. Obviously, for a large state, functional update of the state can become quite expensive. Thus, an ACL2 single-threaded object (stobj) [2] is often used to represent state. The destructive update property of stobjs provides good performance when executing functions on concrete state. The LL2 machine stobj, called simply `s`, contains fields for the Program Counter (PC), local variables, memory, stack, and program storage. All but the first can be thought of as lists. Accessor and updater functions are defined for all fields, with updaters preceded by a '!' character; thus `(loi k s)` retrieves the kth local variable (or register, in LLVM parlance), while `(!loi j val s))` updates the value of the jth register to `val`. Note that `(loi k s)` is defined as `(nth k (rd :locals s))`, and `(!loi j val s)` is defined as `(wr :locals (update-nth j val (rd :locals s)) s)`.

Once the machine state data structure is defined, semantic functions need to be written for all supported instructions. For example, the semantic function for `(EQ x y z)` is as follows:

```
(defun execute-EQ (inst s)
  (declare (xargs :stobjs (s)))
  (let* ((s (!loi (arg1 inst)
                  (if (= (loi (arg2 inst) s) (loi (arg3 inst) s)) 1 0) s))
         (s (!pc (+ 1 (pc s)) s)))
    s))
```

where `inst` is the list form of an instruction (as depicted in Figure 3), `(arg1 inst)` is `(nth 1 inst)`, `(arg2 inst)` is `(nth 2 inst)`, and `(arg3 inst)` is `(nth 3 inst)`. Thus, `execute-EQ` stores the value 1 in the register indicated by the first argument if the value stored in the register indicated by the second argument is equal to the value stored in the register indicated by the third argument; the value 0 is stored in the first argument register otherwise. Finally, the program counter is incremented.

Once semantic functions have been written for every supported instruction, a simple instruction selector function can be composed, as follows:

```
(defun do-inst (inst s)
  (declare (xargs :stobjs (s)))
  (if (equal (op-code inst) 'ADD)
      (execute-ADD inst s)
    (if (equal (op-code inst) 'BR)
        (execute-BR inst s)
      (if (equal (op-code inst) 'CONST)
          (execute-CONST inst s)
          ... s)))...)
```

This instruction selector function is called by the instruction stepper function:

```
(defun step (s)
  (declare (xargs :stobjs (s)))
    (let ((s (do-inst (next-inst s) s)))
      s))
```

where `(next-inst s)` is `(nth (pc s) (program s))`.

Finally, the instruction stepper is called by the top-level LL2 interpeter:

```
(defun ll2 (s n)
  (declare (xargs :stobjs (s)))
  (if (zp n)
      s
      (let* ((s (step s)))
        (ll2 s (- n 1)))))
```

Note that this is all fairly standard technique for defining an instruction set interpreter in ACL2; one peculiarity, however, is that the top-level interpreter argument order (namely, state followed by step count) is mandated by Codewalker.

3.1 Concrete Execution

It is advantageous to be able to validate LLVM programs by running them against concrete inputs. Since all of our interpreter functions are executable, we can readily perform such validation testing. In the ACL2 code of Figure 4, we set up an initial state, establishing an array of length 8 starting at address 100. We write various values into memory at increasing addresses. The array base address is stored in local 0, followed by the `n` and `val` parameters, in locals 1 and 2, respectively. The program is written using the `(wr :program '(...))` form. The program is stepped to conclusion by invoking `(ll2 s 113)`; the return value can be found at `(loi 6 s)`.

As we have written the value 399 into the array three times, when we run the interpreter and fetch the return result as described above, we obtain the expected value: 3. The interpreter executes approximately 226,000 LLVM instructions per second on an ordinary laptop computer. This is approximately one-tenth the speed of our previous method, as is to be expected for an interpreted vs. compiled approach, but this performance level is still more than adequate for validation testing.

4 Codewalker

Now that the interpreter for LL2 is in place, we can begin to use Codewalker to perform decompilation into logic for LLVM programs, producing semantic functions for those programs that the ACL2 user can further reason about. The end goal is to prove that the LLVM code for a given function implements a much more abstract function, written in ACL2, about which we can readily prove interesting correctness properties. In the extensive code documentation for Codewalker, the system is described as follows [18]:

> Two main facilities are provided by Codewalker: the abstraction of a piece of code into an ACL2 “semantic function” that returns the same machine state, and the “projection” of such a function into another function that computes the final value of a given state component using only the values of the relevant initial state components.

```
(include-book "LL2")
(in-package "LL2")

(!loi 0 100 s)
(!loi 1 8 s)
(!loi 2 399 s)
(!memi 100 399 s)
(!memi 101 234 s)
(!memi 102 0 s)
(!memi 103 75 s)
(!memi 104 399 s)
(!memi 105 399 s)
(!memi 106 (1- (expt 2 64)) s)
(!memi 107 20 s)
(!pc 0 s)

(wr :program '((CONST 0)...))

(ll2 s 113)   ;; run to HALT
```

Figure 4: Concrete test case for the occurrences example.

> Codewalker is independent of any particular machine model, as long as a step-based operational semantics for the machine is defined in ACL2. To facilitate this language-independent analysis, the user must declare a "model API" that allows Codewalker to access functionality of the model (e.g., setting the pc in a symbolic state). Generally speaking, Codewalker accesses the model by forming symbolic ACL2 expressions that answer certain questions, then applying the ACL2 simplifier with full access to user-proved lemmas, and then inspecting the resulting term to recover the answer.

Thus, to begin, we tell Codewalker about our operational semantics using `def-model-api`, telling it the name of our interpreter function, the state variable, whether the state is a stobj, the name of the step function, and so on. We next introduce the program to be analyzed, and prove some simple theorems about it, e.g. that writes to state fields other than the program field don't affect the program.

Next, we provide Codewalker with important program-level invariants as well as loop invariants. We also assist the system by providing a measure for the loop clock function, as illustrated in Figure 5.

Finally, we set Codewalker to work, by invoking its `def-semantics` function. First, we ask Codewalker to generate a semantic function for the "preamble" of the code (before the loop), then ask it to produce a semantic function for the loop itself, as shown in Figure 6. We often wish to break up the processing in this way, and not give the entire function to Codewalker in a single chunk. One reason for this is that it can be tricky to craft just the right invariants that are true for preamble, as well as the loop and postlude, and that Codewalker will be able to process successfully.

Codewalker development is still in its early phase, and the system is a bit "touchy" when it comes to the combination of focus regions, invariants, measure annotations, and so on that will result in success. In Codewalker's defense, it is very sophisticated software attempting a very difficult job. To quote the

```
(defun hyps (s)
   (declare (xargs :stobjs (s)))
   (and (sp s)
        (natp (rd :pc s))
        (< (rd :pc s) (len (rd :program s)))
        (< 16 (len (rd :locals s)))
        (integer-listp (rd :locals s))
        (integer-listp (rd :memory s))
        (integer-listp (rd :stack s))))

(defun-nx loop-pc-p (s)
  (= 8 (rd :pc s)))

(defun-nx loop-inv (s)
  (< (nth 5 (rd :locals s))
     (nth 1 (rd :locals s))))

(defun-nx program-inv (s)
  (and (natp (nth 0 (rd :locals s)))
       (natp (nth 1 (rd :locals s)))
       (integerp (nth 2 (rd :locals s)))
       (natp (nth 3 (rd :locals s)))
       (natp (nth 5 (rd :locals s)))
       (natp (nth 6 (rd :locals s)))))

(defun-nx clk-8-measure (s)
  (nfix (if (not (loop-pc-p s))
            (nth 1 (rd :locals s))
          (- (nth 1 (rd :locals s))
             (nth 5 (rd :locals s))))))
```

Figure 5: Some invariants and measures provided to Codewalker.

```
(def-semantics
  :init-pc 0
  :focus-regionp (lambda (pc) (and (<= 0 pc) (< pc 8)))
  :root-name preamble
  :hyps+ ((occurrences-programp s)
          (program-inv s)))

(def-semantics
  :init-pc 8
  :focus-regionp (lambda (pc) (>= pc 8))
  :root-name loop
  :hyps+ ((occurrences-programp s)
          (loop-inv s) (program-inv s)
          (<= (+ (nth 0 (rd :locals s)) (nth 1 (rd :locals s)))
              (len (rd :memory s))))
  :annotations ((clk-loop-8 (declare (xargs :measure (clk-8-measure s))))
                (sem-loop-8 (declare (xargs :measure (clk-8-measure s))))))
```

Figure 6: Invocations of Codewalker `def-semantics` for the occurrences example.

Codewalker documentation: "Def-semantics actually prints a lot of stuff as it goes. It also often fails! Some of its error messages make supposedly helpful suggestions as to what's 'wrong.' Often your response will be to prove more lemmas because things aren't being reduced to the canonical forms. Another response might be to restrict the focus region or strengthen the invariant so as to avoid certain cases." [18]

Codewalker produces decompilations of the indicated code segments, which we can then assemble using functional composition, e.g.:

```
(defun-nx composition (s)
  (sem-loop-8 (sem-preamble-0 s)))
```

Codewalker also produces correctness theorems about the generated semantics functions, e.g.:

```
(DEFTHM SEM-PREAMBLE-0-CORRECT
  (IMPLIES (AND (HYPS S)
                (OCCURRENCES-PROGRAMP S)
                (PROGRAM-INV S)
                (EQUAL (RD :PC S) 0))
           (EQUAL (LL2 S (CLK-PREAMBLE-0 S))
                  (SEM-PREAMBLE-0 S))))

(DEFTHM SEM-LOOP-8-CORRECT
  (IMPLIES (AND (HYPS S)
                (OCCURRENCES-PROGRAMP S)
                (LOOP-INV S)
                (PROGRAM-INV S)
```

```
                (<= (+ (NTH 0 (RD :LOCALS S))
                       (NTH 1 (RD :LOCALS S)))
                    (LEN (RD :MEMORY S)))
                (EQUAL (RD :PC S) 8))
  (EQUAL (LL2 S (CLK-LOOP-8 S))
         (SEM-LOOP-8 S))))
```

The latter theorem states that if the LL2 interpreter is poised at the top of the loop (pc = 8) then running the LL2 interpreter with the occurrences program loaded for a proper number of steps (given by `(CLK-LOOP-8 S)` yields the same result as executing the generated semantic function.

5 Reasoning about LLVM Code via Codewalker Semantic Functions

In order to reason about a function such as `occurrences` in ACL2, we first need to perform abstraction on the data types; particularly, we wish to abstract the input array to a Lisp list. Since we are utilizing stobjs, however, this abstraction has already been provided for us. (Recall that stobjs provide a list abstraction for array data types that feature an efficient, in-place, destructive implementation.)

Next, we need a "golden" list-based specification of `occurrences`. This function should be easy to reason about using ACL2, and so should be written in non-tail-recursive style, as in the following:

```
(defun occurlist (val lst)
  (declare (xargs :guard (and (integerp val) (integer-listp lst))))
  (if (endp lst)
      0
    (+ (if (= val (car lst)) 1 0)
       (occurlist val (cdr lst)))))
```

We wish to prove that the execution of the LLVM instructions of the compiled `occurrences` function operating over an array in memory produces a result equal to the `occurlist` function operating over a list. Unfortunately for the proof of the above, the semantic functions generated by Codewalker are tail-recursive. The proof actually proceeds by the use of two additional functions, a pair of tail-recursive/non-tail-recursive functions that are generated and proved equal by `defiteration`, a book found in `centaur/misc` in the standard ACL2 distribution. (This technique was earlier described in [10].) The call to `defiteration` is as follows:

```
(acl2::defiteration occur-arr (num val s)
  (declare (xargs :stobjs s
                  :guard (and (integerp num) (integerp val))))
  (ifix (+ (if (= (nth ix (rd :memory s)) val) 1 0) num))
   :returns num
   :index ix
   :last (len (rd :memory s)))
```

We first prove that the value stored in the `num_occur` register (register 6) after execution of the composition of semantic functions generated by Codewalker is equal to the result of the tail-recursive function generated by the call to `defiteration` above:

```
(defthm composition-=-occur-arr-tailrec
  (implies
   (and (hyps s)
        (program-inv s)
        (occurrences-programp s)
        (<= (+ (nth 0 (rd :locals s)) (nth 1 (rd :locals s)))
            (len (rd :memory s)))
        (= (nth 1 (rd :locals s)) (len (rd :memory s))))
   (= (nth 6 (rd :locals (sem-loop-8 (sem-preamble-0 s))))
      (occur-arr-tailrec 0 0 (nth 2 (rd :locals s)) s)))
  :hints (("Goal" :in-theory (enable occur-arr-tailrec)
           :cases ((= (len (rd :memory s)) 0) (> (len (rd :memory s)) 0)))))
```

We then prove that the non-tail-recursive function generated by `defiteration` is equal to occurlist:

```
(defthm occur-arr-iter-=-occurlist
  (implies
   (and (sp s) (integerp val) (integer-listp (rd :memory s))
        (= (len (rd :memory s)) (len (rd :memory s))))
   (= (occur-arr-iter (len (rd :memory s)) 0 val s)
      (occurlist val (rd :memory s)))))
```

The above theorem can be proved by first proving the following lemma:

```
(defthm occur-arr-iter-=-occurlist-take--thm
  (implies
   (and
    (sp s) (natp xx) (integerp val)
    (integer-listp (rd :memory s))
    (<= xx (len (rd :memory s))))
   (= (occur-arr-iter xx 0 val s)
      (occurlist val (take xx (rd :memory s)))))
  :hints (("Subgoal *1/1" :in-theory (enable occur-arr-iter))))
```

Since `occur-arr-iter` and `occur-arr-tailrec` are already proved equal by `defiteration`, the proof of `composition-=-occurlist` then follows readily.

```
(defthm composition-=-occurlist
  (implies
   (and (hyps s)
        (program-inv s)
        (occurrences-programp s)
        (<= (+ (nth 0 (rd :locals s)) (nth 1 (rd :locals s)))
            (len (rd :memory s)))
        (= (nth 1 (rd :locals s)) (len (rd :memory s))))
   (= (nth 6 (rd :locals (sem-loop-8 (sem-preamble-0 s))))
      (occurlist (nth 2 (rd :locals s)) (rd :memory s)))))
```

Finally, given the semantic function correctness theorems generated by Codewalker (namely, `SEM-PREAMBLE-0-CORRECT` and `SEM-LOOP-8-CORRECT`, the desired final theorem, depicted in Figure 7, can be stated and proved.

```
(defthm ll2-running-occurrences-code-=-occurlist
  (implies
   (and (hyps s)
        (program-inv s)
        (occurrences-programp s)
        (<= (+ (nth 0 (rd :locals s)) (nth 1 (rd :locals s)))
            (len (rd :memory s)))
        (= (nth 1 (rd :locals s)) (len (rd :memory s)))
        (equal (rd :pc s) 0))
   (= (nth 6 (rd :locals (ll2 (ll2 s (clk-preamble-0 s))
                              (clk-loop-8 (ll2 s (clk-preamble-0 s))))))
      (occurlist (nth 2 (rd :locals s)) (rd :memory s))))
  :hints (("Goal" :cases ((= (len (rd :memory s)) 0)
                          (> (len (rd :memory s)) 0)))
          ("Subgoal 2" :in-theory (enable clk-loop-8))))
```

Figure 7: Final theorem, equating the result of executing the LLVM instructions for the occurrences program to its abstract "golden" specification.

6 Related Work

The technique of compiling to a Virtual Machine instruction set has made a significant comeback in the past twenty years, starting with the JVM, and continuing with Microsoft's CIL, Android Dalvik, and LLVM. Our work on verification at the virtual machine instruction set level was inspired by J Moore's pioneering work on JVM verification [17], as well as Eric Smith's more recent Axe system, which was used to verify a number of Java cryptographic programs at the bytecode level [20].

Zhao *et al.* [23] produced several different formalizations of operational semantics for LLVM in Coq [21], noting that their intention is to produce a verified LLVM compiler, similar to the CompCert verified compiler due to Leroy [13] (CompCert does not utilize the LLVM intermediate form). The goal of Zhao *et al.* was not to produce a verification environment for LLVM bitcode, unlike the present work, but rather to prove the correctness of compiler passes that manipulate LLVM. Jules Villard at Imperial College London is developing llStar, a formal analysis tool for LLVM bitcode. Villard's work so far has focused on proving properties of small LLVM programs that manipulate algebraic data types, utilizing the coreStar symbolic execution engine, separation logic, and SMT technology [22]. LLBMC [7] is a bounded model checker used in bug-finding for C programs that operates on LLVM bitcode. Similary, KITTeL [6] performs termination analysis on C programs by examining LLVM bitcode. Finally, KLEE [3] is a symbolic execution tool that operates on LLVM bitcode to produce coverage test cases and find bugs in C programs.

Codewalker was directly influenced by Magnus Myreen's "decompilation into logic" work [19]. It would be interesting to attempt to replicate the work done here using a combination of Myreen's decompiler and Anthony Fox's L3 instruction set description language [8].

7 Conclusion and Future Work

We have used Codewalker, an instruction-set-neutral decompilation-into-logic system included with the ACL2 theorem prover, to formally analyze C programs that have been compiled to the LLVM intermediate form. Work began by defining a stobj-based interpreter for a subset of the LLVM instruction set, guided by an existing interpreter for the M1 subset of the Java Virtual Machine. Several C programs, including programs to compute factorial, sum of array elements, and number of occurrences of a value in an array, were compiled to LLVM form, and hand-translated to an ACL2-friendly form that could be fed to the interpreter. Validation testing was then conducted on these programs using concrete inputs, before the programs were given to Codewalker for formal analysis. Program-wide invariants, as well as loop invariants and clock measure functions, were defined in order to help Codewalker create semantic functions for program code segments. The composition of these semantic functions was then proved equivalent to more abstract functions: first to a tail-recursive form; then to a non-tail-recursive form (the equality of the latter two having previously been established by the `defiteration` facility); and finally to a top-level non-tail-recursive "golden" specification. Thus, we were able to prove that several sample LLVM programs implement the top-level specifications for those programs.

Future work will focus on using Codewalker to analyze more complex C functions, in particular functions that feature nested loops, as well as functions that employ runtime-allocated memory. We have successfully processed the "straight-line" segments for an LLVM insertion sort program (which features a nested loop) using Codewalker, but have not yet successfully composed the generated semantic functions into a whole program for further analysis. Additionally, now that basic programs operating on unbounded integers have been successfully analyzed using Codewalker, a new version of the LLVM interpreter should be developed that can support different finite data word sizes, as well as the LLVM `call` and `ret` instructions. Finally, we would like to apply Codewalker to additional instruction set architectures, focusing on physical ISAs, as opposed to virtual ISAs like LLVM.

8 Acknowledgments

Many thanks to J Moore for developing Codewalker. I also wish to express my appreciation to J and Matt Kaufmann for several emails that clarified my understanding of Codewalker's capabilities. Thanks also to the anonymous reviewers for their helpful comments. Finally, I wish to acknowledge the wonderful support of my wife, Lori Hardin.

References

[1] Andrew W. Appel (1998): *SSA is Functional Programming*. In: *SIGPLAN Notices*, 33, ACM, pp. 17–20, doi:10.1145/278283.278285.

[2] Robert S. Boyer & J Strother Moore (2002): *Single-Threaded Objects in ACL2*. *PADL 2002*, doi:10.1007/3-540-45587-6_3.

[3] Cristian Cadar, Daniel Dunbar & Dawson Engler (2008): *KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs*. In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, USENIX Association, pp. 209–224. Available at http://dl.acm.org/citation.cfm?id=1855741.1855756.

[4] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman & F. Kenneth Zadeck (1991): *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph*. In: *TOPLAS*, 13, ACM, pp. 451–490, doi:10.1145/115372.115320.

[5] Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Remy & Jerome Vouillon (2014): *The OCaml System Release 4.02 Documentation and Users Guide*. http://caml.inria.fr/distrib/ocaml-4.02/ocaml-4.02-refman.pdf.

[6] Stephan Falke, Deepak Kapur & Carsten Sinz (2011): *Termination Analysis of C Programs Using Compiler Intermediate Languages*. In: *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications (RTA '11)*, pp. 41–50, doi:10.4230/LIPIcs.RTA.2011.41.

[7] Stephan Falke, Florian Merz & Carsten Sinz (2013): *The Bounded Model Checker LLBMC (Tool Demonstration)*. In: *Proceedings of the 28th International Conference on Automated Software Engineering (ASE '13)*.

[8] Anthony Fox (2012): *Directions in ISA Specification*. In: *ITP 2012*, doi:10.1007/978-3-642-32347-8_23.

[9] David S. Hardin, Jennifer A. Davis, David A. Greve & Jedidiah R. McClurg (2014): *Development of a Translator from LLVM to ACL2*. In F. Verbeek & J. Schmaltz, editors: *Proceedings of the 12th International Workshop on the ACL2 Theorem Prover and its Applications*, 152, EPTCS, pp. 163 – 177, doi:10.4204/EPTCS.152.13.

[10] David S. Hardin & Samuel S. Hardin (2013): *ACL2 Meets the GPU: Formalizing a CUDA-based Parallelizable All-Pairs Shortest Path Algorithm in ACL2*. In R. Gamboa & J. Davis, editors: *Proceedings of the 11th International Workshop on the ACL2 Theorem Prover and its Applications*, 114, EPTCS, pp. 127 – 142, doi:10.4204/EPTCS.114.10.

[11] David S. Hardin, Jedidiah R. McClurg & Jennifer A. Davis (2013): *Creating Formally Verified Components for Layered Assurance with an LLVM-to-ACL2 Translator*. In: *Proceedings of the 2013 Layered Assurance Workshop*, ACM.

[12] Matt Kaufmann, Panagiotis Manolios & J Strother Moore (2000): *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, doi:10.1007/978-1-4757-3188-0.

[13] Xavier Leroy (2009): *Formal Verification of a Realistic Compiler*. In: *Communications of the ACM*, 52, pp. 107–115, doi:10.1145/1538788.1538814.

[14] Tim Lindholm, Frank Yellin, Gilad Bracha & Alex Buckley: *The Java Virtual Machine Specification, Java SE 8 Edition*. https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf.

[15] LispWorks Ltd.: *Common Lisp HyperSpec*. http://www.lispworks.com/documentation/HyperSpec/Front/index.htm.

[16] LLVM Project: *The LLVM Compiler Infrastructure*. http://llvm.org/.

[17] J Strother Moore (1999): *Proving Theorems about Java-like Byte Code*. In E.-R. Olderog & B. Steffen, editors: *Correct System Design — Recent Insights and Advances, Lecture Notes in Computer Science* 1710, Springer-Verlag, pp. 139–162, doi:10.1007/3-540-48092-7_7.

[18] J Strother Moore (2014): *Codewalker source code*. Standard ACL2 distribution at http://www.cs.utexas.edu/users/moore/acl2.

[19] Magnus O. Myreen, Michael J. C. Gordon & Konrad L. Slind (2012): *Decompilation into Logic — Improved*. In: *FMCAD'12*, ACM/IEEE.

[20] Eric Smith (2011): *Axe: An Automated Formal Equivalence Checking Tool for Programs*. Ph.D. thesis, Stanford University.

[21] The Coq Development Team (2015): *The Coq Proof Assistant Reference Manual, Version 8.4pl6*. https://coq.inria.fr/distrib/current/files/Reference-Manual.pdf.

[22] Jules Villard (2013): *Here be wyverns! Verifying LLVM bitcode with llStar*. Unpublished manuscript at http://www.doc.ic.ac.uk/~jvillar1/pub/llstar-draft-oct13.pdf.

[23] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin & Steve Zdancewic (2012): *Formalizing the LLVM Intermediate Representation for Verified Program Transformations*. In: *POPL'12*, ACM, doi:10.1145/2103621.2103709.

Stateman: Using Metafunctions to Manage Large Terms Representing Machine States

J Strother Moore
Department of Computer Science
The University of Texas at Austin
moore@cs.utexas.edu *

When ACL2 is used to model the operational semantics of computing machines, machine states are typically represented by terms recording the contents of the state components. When models are realistic and are stepped through thousands of machine cycles, these terms can grow quite large and the cost of simplifying them on each step grows. In this paper we describe an ACL2 book that uses HIDE and metafunctions to facilitate the management of large terms representing such states. Because the metafunctions for each state component updater are solely responsible for creating state expressions (i.e., "writing") and the metafunctions for each state component accessor are solely responsible for extracting values (i.e., "reading") from such state expressions, they can maintain their own normal form, use HIDE to prevent other parts of ACL2 from inspecting them, and use honsing to uniquely represent state expressions. The last feature makes it possible to memoize the metafunctions, which can improve proof performance in some machine models. This paper describes a general-purpose ACL2 book modeling a byte-addressed memory supporting "mixed" reads and writes. By "mixed" we mean that reads need not correspond (in address or number of bytes) with writes. Verified metafunctions simplify such "read-over-write" expressions while hiding the potentially large state expression. A key utility is a function that determines an upper bound on the value of a symbolic arithmetic expression, which plays a role in resolving writes to addresses given by symbolic expressions. We also report on a preliminary experiment with the book, which involves the production of states containing several million function calls.

1 Background

ACL2 [3, 2] is frequently used to model computing machines via operational semantics. It is not difficult to configure the ACL2 theorem prover so that it can use the definitions of the machine semantics and a few well-chosen rewrite rules to step through code sequences, split on tests, induct on loops, etc. Examples of these methods being used to prove functional correctness of code under formal operational semantics may be found in numerous publications [6, 7, 10, 1]. Such symbolic state terms can grow quite large when many steps are composed. The question addressed here is: *how can we exploit ACL2's rewriter to symbolically execute formalized code while preventing it from slowing down as state expressions get large?*

This paper describes the Stateman book for managing large terms representing machine states in ACL2 models of computing machines. "Stateman" stands for "state management." This is a work in progress and this paper has many brief descriptions of intended **Future Work**.

The idealistic dream is that a user wishing to model some byte-addressed computing machine and do code proofs or run the Codewalker tool[1] might build the operational semantics on top of the state

*This work was partially supported by ForrestHunt, Inc.

[1]Codewalker extracts ACL2 functions from machine code given the formal operational semantics of the ISA and is sim-

M. Kaufmann and D. Rager (Eds.): ACL2 Workshop 2015 (ACL2 2015).
EPTCS 192, 2015, pp. 93–109, doi:10.4204/EPTCS.192.8

provided by Stateman and thereby inherit the state management techniques here described. But machine models are very idiosyncratic. Users may actually need to design their own states and merely exploit the basic techniques described here. Thus, this paper focuses mainly on the design decisions in our work. As usual, readers are welcome, indeed encouraged, to read the Stateman book itself and use it as the basis of their own versions.

We start with a brief description of our generic state, then we present the highlights of our state management techniques, provide some examples, discuss a few details, and present some preliminary performance measures.

2 The Generic State

The book provides a generic single-threaded object, `ST` (henceforth, *st*), providing three fields. See :DOC stobj.[2]

```
(defstobj st
  (I  :type unsigned-byte :initially 0)          ; program counter
  (S  :initially nil)                            ; status
  (M  :type (array (unsigned-byte 8) (*m-size*)) ; memory
      :initially 0
      :resizable nil
      )
  :inline t
  :renaming
  ((UPDATE-I   !I)
   (UPDATE-S   !S)
   (UPDATE-MI  !MI)
   (M-LENGTH   ML)))
```

The primitive accessors are `I`, `S`, and `MI`, and the primitive updaters are `!I`, `!S`, and `!MI`.[3] The `I` and `S` fields were originally intended for the machine's instruction counter and status flag, and `MI` provides a byte addressed memory of 8-bit bytes. The person using this book to model the state of a computing machine need not use the `I` and `S` fields for their implied purposes. The modeler might, for instance, choose to store all state information including the instruction counter and various status bits in the byte addressed memory and ignore the `I` and `S` fields altogether.

Byte-addresses are integers starting at 0. The byte-addressed memory is of fixed size, `*m-size*`, which is currently only `5312`. This constant is a holdover from the earliest use of the state and (**Future Work**) will be generalized in future work. Indeed, the whole development would have been easier were there no upper bound on memory size. Imposing an upper bound forced certain issues to be dealt with –

ilar to the HOL decompilation work by Magnus Myreen[8, 9]. See the `README` file in the Community Book directory `projects/codewalker/`. The version of Codewalker used here is still experimental.

[2] When we say "See :DOC *x*" we mean see the documentation topic *x* in the ACL2 documentation, which may be found by visiting the ACL2 home page[4], clicking on The User's Manuals, then clicking on the ACL2+Books Manual and typing *x* into the "Jump to" box.

[3] The third field of the single-threaded object is named `M` and is an array, but only the elements can be accessed or changed, with `MI` and `!MI`.

issues that are necessarily raised in any realistic model. The magnitude of that upper bound is practically irrelevant from the research perspective.

The Stateman book uses `MI` and `!MI` only to provide support for two more general utilities, `R` and `!R`, for reading and writing an arbitrary number of bytes. We do not think of `MI` and `!MI` as "visible" to the user of Stateman.

It is best to think of the generic state as providing the following functionality:

expression	value
`(I` *st*`)`	instruction counter of state *st*
`(S` *st*`)`	status flag of state *st*
`(R` *a n st*`)`	natural number obtained by reading *n* bytes starting at address *a* in the memory of state *st*
`(!I` *v st*`)`	new state obtained from state *st* by setting the instruction counter to *v*
`(!S` *v st*`)`	new state obtained from state *st* by setting the status flag to *v*
`(!R` *a n v st*`)`	new state obtained by writing *n* bytes of natural number *v* into the memory of *st* starting at address *a*

`R` and `!R` use the "Little Endian" convention. For example, `(!R` *a n v st*`)` writes the less significant bytes of *v* to the lower addresses, with the least significant byte written to address *a* and all other bytes written to larger addresses. (**Future Work**) We would like to support either Little or Big Endian conventions.

Nests of `!I`, `!S`, and `!R` applications are called *state expressions* or *state terms* because they denote machine states. Any term whose top function symbol is `I`, `S`, or `R` applied to a state expression is called a *read-over-write* expression. Any term whose top function symbol is `!I`, `!S`, or `!R` applied to a state expression is called a *write-over-write* expression. Of course, write-over-write expressions are themselves state expressions.

Our concern here is simplifying read-over-write and write-over-write expressions in support of code proofs and code walks. These issues are straightforwardly managed with rewrite rules. For example, the read over write expression `(R 24 8 (!R 40 8` *v st*`))` can be simplified to `(R 24 8` *st*`)`. But as state expressions grow large – and they can grow very large when long code sequences are involved – two problems crop up.

First, the rewriter tends to re-simplify parts of states that have already been simplified. Second, the traditional rewrite rules for handling byte-addressed memory involve backchaining to establish that byte sequences do not overlap. For example, the rewrite rules that replace `(R` *a n* `(!R` *b k v st*`))` by `(R` *a n st*`)` have the hypotheses `(natp` *a*`)`, `(natp` *b*`)`, `(natp` *n*`)`, `(natp` *k*`)`, and either `(<` `(+` *a n*`)` *b*`)` or `(<` `(+` *b k*`)` *a*`)`. The inequalities can get very expensive when *a* and *b* are large arithmetic expressions. Furthermore, *a* and *b* typically become large arithmetic expressions when the code being explored is doing indexed addressing (as in array access) and long code sequences are involved in the computation of the indices. Every read-over-write and write-over-write expression raises such an *overlap* question. Furthermore, a read of a deeply nested state expression typically raises an overlap question for each write in the nest. For speed we must answer overlap questions without resorting to heavy-duty arithmetic.

3 Highlights of Key Design Decisions

Some of the key decisions in the design of Stateman are listed and briefly elaborated below. In the next section, where we give examples, we discuss the implications of some of these decisions.

- **Manage read-over-write and write-over-write expressions exclusively with metafunctions**: Stateman defines a metafunction for each of `I`, `S`, `R`, `!I`, `!S`, and `!R`. These metafunctions are named `meta-I`, `meta-S`, etc. Like all metafunctions, they take terms as input and yield possibly different terms as output.[4] The metafunctions for `R` and `!R` are extended metafunctions and thus additionally take the metafunction context and ACL2 state as arguments. These two metafunctions only use the type-alist in the metafunction context and they ignore the ACL2 state. However, the biggest problem faced by these functions is the read-over-write overlap questions: "is one address less than another?", given only the syntactic expressions representing the two addresses. This motivates the next item.
- **Implement a syntactic interval inference mechanism**: Imagine a function that when given an arithmetic/logical term, can infer an upper bound. This is quite different functionality than normally found in ACL2. ACL2 can be configured to answer questions like "Is α less than 16?" but here we want a utility for answering "What number is α less than?" This functionality is especially important in codewalking unknown code. Suppose the code in question uses α as an index into some array at location *base*. What part of the state is changed if the code writes to $base + \alpha$? If you know enough about the code to know the bound on the array, you could undertake to prove that α is in bounds and thus conclude that only the array is affected by the write. But if you do not know much about the code, you need an inference mechanism to deduce a bound on α. Stateman provides a verified interval inference mechanism named `Ainni` which is discussed in more detail in Section 5.
- **Implement syntactic means of deciding some inequalities**: Given `Ainni`, it is possible to implement the extended metafunction `meta-<` that takes an inequality and the metafunction context and decides many inequalities, `(< `α β`)`, by computing intervals for α and β and comparing their endpoints, e.g., if the upper bound of α is below the lower bound of β, then the inequality is true. This can save backchaining into linear arithmetic on large arithmetic/logical expressions.
- **Implement syntactic means of simplifying some `MOD` expressions**: In machine arithmetic, expressions of the form `(MOD `α *'n*`)` frequently arise, where `n` is some natural number. Some expressions of this sort can be simplified by syntactic means given the ability to infer bounds on α. See Section 6.
- **Use syntactic means to decide overlap questions**: Suppose the type-alist tells us that the 32-bit word at address 8, i.e., `(R 8 4 `*st*`)` is less than 16. Then a quick syntactic scan of the address expression `(+ 3200 (* 8 (R 8 4 `*st*`)))` reveals that the value lies in the interval [3200, 3320] and so reading, say, 3 bytes from that address might touch any address in the interval [3200, 3322].
- **Insist that all byte counts be quoted constants**: This facilitates the interval analysis mentioned above. We do not regard it as a restriction given Stateman's intended application for code analysis. In most ISAs the number of bytes to be manipulated by an instruction is explicitly given in the instruction or else is fixed by the instruction or the architecture.

[4] Metafunctions traffic in fully translated terms but the examples in this paper generally show untranslated terms for readability.

- **Do not put nested `!R`-expressions into address order**: We leave the most recent writes at the top of the state expression under the assumption that program code tends to read from addresses recently written.
- **Eliminate perfectly shadowed writes**: When `!R`, with address *a* and byte count *n*, is applied to a state expression already containing an application of `!R` with address *a* with byte count *n*, Stateman eliminates the inner (earlier) one. Similar considerations apply to nested `!I` and `!S` calls. This reduces the size of the final state expression. But Stateman does not try to eliminate partially shadowed writes. We explain below.
- **Use `hons` rather than `cons` to create state expressions**: This means that if the same state expression is created along different paths of a code proof or walk, no additional space is allocated; furthermore, hons facilitates the use of memoization.
- `HIDE` **the state expressions produced by the metafunctions**: This ensures that no rewrite rule touches them. For example, if a machine model mentions an expression like

```
(!R 32 4 v
    (!R 8 4 (+ (R 8 4 st) 4)
        (!I 123
            (!S NIL st))))
```

  as would happen if it set the status flag to `NIL`, the instruction pointer to `123`, incremented the word at address 8 by 4 and wrote *v* to the word at address 32, then the inside-out rewriting of ACL2 would invoke the metafunctions for `!S`, then `!I`, and then `!R` (twice) and ripple a `HIDE` out so the final term would be as exactly as above but with a single `HIDE` around it at the top level. It would never be further simplified except by these metafunctions.
- `HIDE` **some values extracted by reads from hidden states to avoid re-simplifying them**: This is a controversial decision and is still quite unsettled. (**Future Work**) The issue is that over long codewalks (involving thousands of instructions) the expressions built up as values in the memory can be huge. By embedding extracted values in `HIDE` expressions, they are not re-simplified. The downside is that it can be impossible to decide simple tests because one does not know much about the hidden expressions. A compromise would be to bury the `HIDE`s several levels down in the extracted expressions, leaving the top few function symbols available. At the moment, all extracted values are hidden except constants and calls of `R`. This means that the metafunctions here must remove some `HIDE`s from values before storing them into memory.
- **Prove guards and well-formedness guarantees of the metafunctions**: ACL2 users should be well aware of the efficiency advantages of verifying the guards on functions used in heavy-duty computations. A less familiar topic, though, is discussed in the new feature documented in :DOC well-formedness-guarantee. It has long been the case that when a metafunction is applied the theorem prover checks that the result is a well-formed term, by running the function `termp` on the output and the current ACL2 world. This hidden cost of metafunctions goes all the way back to the origin of ACL2 in 1989. However, when the output of a metafunction is huge, the well-formedness check can be expensive, and the basic supposition in the Stateman work is that state expressions are huge. A new feature of ACL2 Version 7.2 makes it possible to skip the well-formedness check by *proving* that the metafunction always returns a `termp`. We have found that providing such well-formedness guarantees is worthwhile in Stateman. See [5]. We give some data on this below.

4 Examples

We illustrate these ideas with a few examples. The reader may notice two odd aspects to our examples. The first is that most addresses illustrated are quoted constants. The second is that when non-constant expressions occur as addresses the only variable involved is *st* and it always occurs in a primitive state accessor like `(R` *a n st*`)`. We do not believe these are serious constraints if Stateman is used for code analysis: Typical code, especially binary machine code, refers to fixed addresses or offsets from other addresses (as in array indexing and stack slots relative to some stack or frame pointer in a register); "variables" are just the contents of memory locations at such addresses. However (**Future Work**) it would not be difficult to support variable symbols provided the context established natural number bounds on their values.

Examples (1)–(7) below are extracted verbatim from a session log that started in a fresh ACL2 with the inclusion of the Stateman book. Because this is a work in progress, we keep the version number as part of the book name right now. This log started by including `stateman22.lisp` which is included in the supplemental material. The supplemental material also includes `simple-examples.lsp`, a file (not a book) showing the actual input forms for these and some other examples in this paper. We hope those forms can help the user who wishes to extend Stateman's functionality.

```
ACL2 !>(meta-!I '(!I '123 st))                                          ;(1)
(HIDE (!I '123 ST))

ACL2 !>(meta-!R '(!R '0 '4 (R '16 '4 st) (HIDE (!I '123 ST)))           ;(2)
                nil state)
(HIDE (!R '0 '4 (R '16 '4 ST) (!I '123 ST)))                            ;(st')

ACL2 !>(meta-I '(I (HIDE (!R '0 '4 (R '16 '4 st)  (!I '123 ST)))))     ;(3)
'123

ACL2 !>(meta-R '(R '0 '4 (HIDE (!R '0 '4 (R '16 '4 st) (!I '123 ST)))) ;(4)
               nil state)
(R '16 '4 ST)

ACL2 !>(meta-R '(R '2 '2 (HIDE (!R '0 '4 (R '16 '4 st) (!I '123 ST)))) ;(5)
               nil state)
(HIDE (ASH (R '16 '4 ST) '-16))

ACL2 !>(meta-R '(R '8 '4 (HIDE (!R '0 '4 (R '16 '4 st) (!I '123 ST)))) ;(6)
               nil state)
(R '8 '4 ST)

ACL2 !>(meta-R '(R '2 '4 (HIDE (!R '0 '4 (R '16 '4 st) (!I '123 ST)))) ;(7)
               nil state)
(HIDE (BINARY-+ (ASH (R '4 '2 ST) '16)
                (ASH (R '16 '4 ST) '-16)))
```

In example (1) we call the metafunction for `!I` on the term `(!I '123 st)`, just as the rewriter does when it encounters a `!I`-term. The result is a hidden state. Notice that metafunctions traffic in fully translated terms.

In example (2) we call the metafunction for `!R` on the `!R`-term that writes the 4-byte value of `(R '16 '4 ST)` to location `0` in the previously produced (now hidden) state. Note that the metafunction for `!R` takes two additional arguments, the metafunction context, in this case `nil`, and the ACL2 state object, since `meta-!R` is an extended metafunction. Again, nothing significant happens except the new state is hidden. Henceforth in this narrative we will refer to the state produced by (2) as st'.

In example (3) we use the metafunction for `I` to extract the instruction counter of st'.

In example (4) we use the metafunction for `R` to read (4 bytes of) the contents of address 0 in st'. The result is exactly what was written in (2) because it was 4 bytes long.

In example (5) we read the last two bytes of that previously written quantity, that is, we read 2 bytes starting at address 2 in st'. Two things are noteworthy. One is that it is reported as the 4-byte quantity that was written in (2), shifted down by 16 bits. The second is that it is hidden – the "controversial" decision.

In example (6) we read from an address above any affected by the write in st'. The result is whatever was there in the original state *st*.

In example (7) we read 4 bytes starting at address 2 in st'. This is a "mixed" read in the sense that the result involves the last two bytes from what was written at address 0 and the bytes that were at locations 4 and 5 of the original state *st*. It is expressed as a sum, with the latter bytes shifted up. Again, it is (controversially) hidden.

It is important to realize that all of these transformations are carried out by verified metafunctions without involving rewrite rules, linear arithmetic, or other heavy-duty theorem proving. Consequently, these transformations are very fast.

Since the `I` and `S` slots are unaffected by writes to memory and do not involve addresses or overlap issues our examples below focus on `R`- and `!R`-terms.

Henceforth, we will display untranslated terms for both input and output and will not exhibit the calls of the relevant metafunction. Instead, the reader should understand that the notation "$\alpha \Longrightarrow \beta$" means that α is transformed to β by the metafunction appropriate for the top function symbol of α. Since both `meta-R` and `meta-!R` take a metafunction context we make clear in the surrounding narrative what the context is. This only involves describing the governing assumptions (as encoded in the type-alist). Finally, instead of writing something like "$\alpha \Longrightarrow$ `(IF` *hyp* β α`)`" we will generally write "$\alpha \Longrightarrow^{\dagger} \beta$" and describe the side condition *hyp* generated by the metafunction in the accompanying narrative. Recall that before such an α is replaced by β the rewriter must establish *hyp*.

Given a metafunction context in which the type-alist is empty, we can thus recap lines (1)–(7) above with:

```
(!I 123 st)                                                    ;(1)
⟹
(HIDE (!I 123 st))

(!R 0 4 (R 16 4 st) (HIDE (!I 123 st)))                        ;(2)
⟹
(HIDE (!R 0 4 (R 16 4 st) (!I 123 st)))

(I (HIDE (!R 0 4 (R 16 4 st)  (!I 123 st))))                   ;(3)
⟹
```

```
123
```

```
(R 0 4 (HIDE (!R 0 4 (R 16 4 st) (!I 123 st))))                    ;(4)
⟹
(R 16 4 st)

(R 2 2 (HIDE (!R 0 4 (R 16 4 st) (!I 123 st))))                    ;(5)
⟹
(HIDE (ASH (R 16 4 st) -16))

(R 8 4 (HIDE (!R 0 4 (R 16 4 st) (!I 123 st))))                    ;(6)
⟹
(R 8 4 st)

(R 2 4 (HIDE (!R 0 4 (R 16 4 st) (!I 123 st))))                    ;(7)
⟹
(HIDE (+ (ASH (R 4 2 st) 16)
         (ASH (R 16 4 st) -16)))
```

Relatively little work is done on simplifying writes, aside from looking for shadowed writes to be deleted. For example, one might wonder at the simple

```
(!R 8 4 v st)                                                      ;(8)
⟹
(HIDE (!R 8 4 v st))
```

since *v* might be too big to fit in 4 bytes. But instead of truncating *v* on write we do so on read:

```
(R 8 4 (HIDE (!R 8 4 v st)))                                       ;(9)
⟹
(HIDE (MOD (IFIX v) 4294967296))
```

Now let the metafunction context encode the assumption that `(R 16 4 st)` is less than 16. In the example below, we treat `(R 16 4 st)` as an index into a QuadWord array (8-byte per entry) based at address 3200.

```
(R (+ 3200 (* 8 (R 16 4 st))) 8                                    ;(10)
   (HIDE (!R 3600 4 v (!R 8 4 w st))))
⟹†
(R (+ 3200 (* 8 (R 16 4 st))) 8 st)


(!R (+ 3200 (* 8 (R 16 4 st))) 8 u                                 ;(11)
    (HIDE (!R 3600 4 v
              (!R 8 4 w
                  (!R (+ 3200 (* 8 (R 16 4 st))) 8 x
                      st)))))
```

$\Longrightarrow$

```
(HIDE (!R (+ 3200 (* 8 (R 16 4 st))) 8 u
          (!R 3600 4 v
              (!R 8 4 w st))))
```

The "†" on the transformation in (10) indicates that a side condition was generated. That side condition is `(<= (R 16 4 st) 15)`, and it must be established before the replacement is made. Establishing such side conditions should be trivial since they are extracted from the type-alist in the metafunction context. Given that condition, we see that the 8-byte read at `(+ 3200 (* 8 (R 16 4 st)))` may only touch bytes in the interval [3200, 3327]. We discuss this interval analysis further below. But because of it, the metafunction can determine that neither of the two writes in the hidden state of (10) is relevant since the 4 bytes starting at 3600 are above the target interval and 4 bytes starting at 8 are below it.

Interestingly, no side condition is necessary on transformation (11). If `(R 16 4 st)` is sufficiently large the new write at `(+ 3200 (* 8 (R 16 4 st)))` *might* shadow out the write at 3600, but that does not matter because the new write is added at the top of the expression (chronologically after the write at 3600), so the answer above is correct. And, regardless of the magnitude of `(R 16 4 st)`, the new write shadows out the earlier one at the exact same address and the earlier write can be dropped.

Our final example is contrived to show a mixed read that spans several chronologically separated writes. The empty metafunction context is sufficient for this example. We will ultimately read 8 bytes starting at address 3. But consider the writes that create the relevant memory. The write of 4 bytes of *v* at address 2 is partially shadowed by the write of 4 bytes of *u* at address 0. The writes at 14 and 19 are irrelevant because we only need bytes 3 through 10. The first byte of our answer is the high order byte of *u* written at address 3. The next two bytes are the two high order bytes of *v* at addresses 4 and 5. Then we get 3 bytes from the original *st* at addresses 6, 7, and 8, and finally we get the two low order bytes from *w* at addresses 9 and 10. We then assemble these 8 bytes using the Little Endian notation and put the final sum into ACL2's term order.

```
(R 3 8                                                          ;(12)
   (HIDE
    (!R 14 5 x
        (!R 0 4 u
            (!R 19 8 y
                (!R 9 2 w
                    (!R 2 4 v st)))))))
```

$\Longrightarrow$

```
(HIDE (+ (ASH (R 6 3 st) 24)
         (+ (MOD (ASH (IFIX u) -24) 256)
            (+ (ASH (MOD (IFIX w) 65536) 48)
               (ASH (MOD (ASH (IFIX v) -16) 65536) 8)))))
```

(**Future Work**) We are dissatisfied with the normal form of expressions denoting the results of mixed reads. To be more precise, we do not have enough experience with it yet to know whether it is sufficient for our purposes. The current implementation uses `IFIX` to convert terms to integer form as required by basic rules for `ASH` (if syntactic analysis cannot establish that the term returns an integer), uses `MOD` to truncate unneeded higher order bits, and uses `ASH` to shift bits into the right locations. The question however is this: Suppose such an expression is written to a memory location and then one must read a few bytes from it. The current metafunctions produce `ASH`/`MOD`-terms that could be further simplified.

But given the controversial decision to HIDE the complicated results of reads, that simplification should be done inside meta-R.

Stateman does not produce normalized states for at least two reasons. First, it does not put writes into address order. Second it does not eliminate partial shadows. Why bother to eliminate partially shadowed material if one can read out the answers if and when needed? This consideration is especially relevant since resolving a partial shadow generally makes the state syntactically *larger*, e.g., to resolve the shadowing of the write at 2 above one would replace (!R 2 4 *v st*) by the larger term (!R 4 2 (ASH (IFIX *v* -16)) *st*). It is not clear this is an improvement. Furthermore, we suspect partial shadowing is fairly rare compared to "perfect shadowing" where the *n* bytes starting at address *a* are repeatedly reused for different *n* byte values.

(**Future Work**) But the lack of normalization raises the question of determining state equality. Stateman does not support state equality at the moment. But the plan is to support it by a metafunction that announces the equality of two states formed by different sequences of writes to the same initial state by checking that every read of every byte written to either state produces the same expression.

5 Ainni: Abstract Interpreter for Natural Number Intervals

Perhaps the most important idea to come out of this work so far is the development and verification of an ACL2 function that takes the quotation of a term together with a type-alist and attempts to determine a closed natural number interval containing the value of the term. This function is called Ainni, which stands for *Abstract Interpreter for Natural Number Intervals*. Ainni can be thought of as a "type-inference" mechanism for a class of ACL2 arithmetic expressions, except the "types" it deals with are intervals over the naturals.

Ainni explores terms composed of constants, the state *st*, and the function symbols +, -, *, R, HIDE, MOD, ASH, LOGAND, LOGIOR, and LOGXOR.[5] (**Future Work**) This set of function symbols was determined by seeing what functions were introduced by the codewalk of a particularly large and challenging test program: an implementation of DES. Essentially, Ainni should support all of the basic functions used in the semantics of the ALU operations of the machine being formalized. We therefore anticipate that the list here will have to grow.

Ainni recursively descends through the term "evaluating" the arguments of function calls – only in this case that means computing intervals for them – and then applying bounders (see the discussion of "bounders" in :DOC tau-system) corresponding to the function symbols to obtain an interval containing all possible values of the function call. At the bottom, which in this case are calls of R, Ainni uses the type-alist to try to find bounds on reads that are tighter than the syntactically apparent $0 \leq$ (R *a n st*) $\leq 2^{8n} - 1$. (**Future Work**) It is here, at the "bottom" of the recursion, that we could add support for variable symbols or unknown function symbols.

For example, consider the quotation of the term

```
(+ 288 (* 8 (LOGAND 31 (ASH (R 4520 8 st) -3)))).
```

In the absence of any contextual information, Ainni returns the natural number interval [288,536]. The reasoning is straightforward: we know that (R 4520 8 *st*) is a natural in the interval $[0, 2^{64} - 1]$. The tau-bounder for ASH tells us that shifting it right 3 reduces that to $[0, 2^{61} - 1]$, and then the tau-bounder for LOGAND tells us that bitwise conjoining it with 31 shrinks the interval to [0,31]. Multiplying by 8 makes the interval [0, 248], and adding 288 makes it [288, 536].

[5] Several of these symbols are macros that expand into calls of function symbols that Ainni actually recognizes.

By default `(R 4520 8 st)` is known to lie in $[0,2^{64}-1]$, but the type-alist might restrict it to a smaller interval. For example, it might assert that `(R 4520 8 st)` < 24, in which case `Ainni` determines that the term above lies in the interval [288,304].

In addition to returning the interval, `Ainni` also returns a flag indicating whether the term was one that `Ainni` could confine to a bounded natural interval and a list of hypotheses that must be true for its interval to be correct. These hypotheses have two sources: (i) assumptions extracted from the context and (ii) `Ainni`'s inherent assumptions (such as a built-in assumption that no computed value is negative[6], which might translate to the hypothesis `(not (< x y))` if the term is `(- x y)`).

Finally, `Ainni` is verified to be correct. That is, the certification of Stateman involves a proof of the formal version of:

> Let x be the quotation of an ACL2 term and ta be a type-alist. Let flg, $(h_1 \ \ldots \ h_k)$ and $[lo, hi]$ be the flag, hypotheses, and the interval returned by `Ainni` on x and ta. Then if flg is true:
>
> - $(h_1 \ \ldots \ h_k)$ is a list of quotations of terms,
> - lo and hi are natural numbers such that $lo \leq hi$, and
> - if $(\mathscr{E} \ h_i \ a) =$ `T` for each $1 \leq i \leq k$, then $lo \leq (\mathscr{E} \ x \ a) \leq hi$, where $\mathscr{E}$ is an evaluator that recognizes the function symbols handled by `Ainni`.

`Ainni` is used in `meta-R` to handle the overlap questions that arise. In addition, it is used in `meta-<` to decide some inequalities and in `meta-MOD` to simplify some `MOD` expressions.

Furthermore, `Ainni` is fast. For example, in the codewalk of the DES algorithm, one particular index expression is a nest of 382 function calls containing every one of the function symbols known to `Ainni`. Just for fun, here is the expression, printed "almost flat" (without much prettyprinting):

```
(LOGIOR
  (LOGAND 32 (ASH (MOD (ASH (LOGXOR (LOGIOR (ASH (ASH (LOGIOR (ASH (MOD (ASH (R 4520 8 ST) 0) 2) 31) (ASH (LOGAND 4026531840
    (R 4520 8 ST)) -1) (ASH (MOD (ASH (R 4520 8 ST) -27) 2) 26) (ASH (MOD (ASH (R 4520 8 ST) -28) 2) 25) (ASH (LOGAND 251658240 (R 4520
    8 ST)) -3) (ASH (MOD (ASH (R 4520 8 ST) -23) 2) 20) (ASH (MOD (ASH (R 4520 8 ST) -24) 2) 19) (ASH (LOGAND 15728640 (R 4520 8 ST))
    -5) (ASH (MOD (ASH (R 4520 8 ST) -19) 2) 14) (ASH (MOD (ASH (R 4520 8 ST) -20) 2) 13) (ASH (LOGAND 983040 (R 4520 8 ST)) -7) (ASH
    (MOD (ASH (R 4520 8 ST) -15) 2) 8)) -8) 24) (ASH (LOGIOR (ASH (MOD (ASH (R 4520 8 ST) -16) 2) 31) (ASH (LOGAND 61440 (R 4520 8 ST))
    15) (ASH (MOD (ASH (R 4520 8 ST) -11) 2) 26) (ASH (MOD (ASH (R 4520 8 ST) -12) 2) 25) (ASH (LOGAND 3840 (R 4520 8 ST)) 13) (ASH (MOD
    (ASH (R 4520 8 ST) -7) 2) 20) (ASH (MOD (ASH (R 4520 8 ST) -8) 2) 19) (ASH (LOGAND 240 (R 4520 8 ST)) 11) (ASH (MOD (ASH (R 4520 8
    ST) -3) 2) 14) (ASH (MOD (ASH (R 4520 8 ST) -4) 2) 13) (ASH (MOD (R 4520 8 ST) 16) 9) (ASH (ASH (R 4520 8 ST) -31) 8)) -8)) (R (+
    4376 (* 8 (R 4536 8 ST)) (* 8 (- (R 4528 8 ST)))) 8 ST)) -40) 256) -2))
  (ASH (MOD (ASH (MOD (ASH (LOGXOR (LOGIOR (ASH (ASH (LOGIOR (ASH (MOD (ASH (R 4520 8 ST) 0) 2) 31) (ASH (LOGAND 4026531840 (R 4520
    8 ST)) -1) (ASH (MOD (ASH (R 4520 8 ST) -27) 2) 26) (ASH (MOD (ASH (R 4520 8 ST) -28) 2) 25) (ASH (LOGAND 251658240 (R 4520 8 ST))
    -3) (ASH (MOD (ASH (R 4520 8 ST) -23) 2) 20) (ASH (MOD (ASH (R 4520 8 ST) -24) 2) 19) (ASH (LOGAND 15728640 (R 4520 8 ST)) -5) (ASH
    (MOD (ASH (R 4520 8 ST) -19) 2) 14) (ASH (MOD (ASH (R 4520 8 ST) -20) 2) 13) (ASH (LOGAND 983040 (R 4520 8 ST)) -7) (ASH (MOD (ASH
    (R 4520 8 ST) -15) 2) 8)) -8) 24) (ASH (LOGIOR (ASH (MOD (ASH (R 4520 8 ST) -16) 2) 31) (ASH (LOGAND 61440 (R 4520 8 ST)) 15) (ASH
    (MOD (ASH (R 4520 8 ST) -11) 2) 26) (ASH (MOD (ASH (R 4520 8 ST) -12) 2) 25) (ASH (LOGAND 3840 (R 4520 8 ST)) 13) (ASH (MOD (ASH (R
    4520 8 ST) -7) 2) 20) (ASH (MOD (ASH (R 4520 8 ST) -8) 2) 19) (ASH (LOGAND 240 (R 4520 8 ST)) 11) (ASH (MOD (ASH (R 4520 8 ST) -3)
    2) 14) (ASH (MOD (ASH (R 4520 8 ST) -4) 2) 13) (ASH (MOD (R 4520 8 ST) 16) 9) (ASH (ASH (R 4520 8 ST) -31) 8)) -8)) (R (+ 4376 (* 8
    (R 4536 8 ST)) (* 8 (- (R 4528 8 ST)))) 8 ST)) -40) 256) -2) 32) -1)
  (ASH (MOD (ASH (MOD (ASH (LOGXOR (LOGIOR (ASH (ASH (LOGIOR (ASH (MOD (ASH (R 4520 8 ST) 0) 2) 31) (ASH (LOGAND 4026531840 (R 4520
    8 ST)) -1) (ASH (MOD (ASH (R 4520 8 ST) -27) 2) 26) (ASH (MOD (ASH (R 4520 8 ST) -28) 2) 25) (ASH (LOGAND 251658240 (R 4520 8 ST))
    -3) (ASH (MOD (ASH (R 4520 8 ST) -23) 2) 20) (ASH (MOD (ASH (R 4520 8 ST) -24) 2) 19) (ASH (LOGAND 15728640 (R 4520 8 ST)) -5) (ASH
    (MOD (ASH (R 4520 8 ST) -19) 2) 14) (ASH (MOD (ASH (R 4520 8 ST) -20) 2) 13) (ASH (LOGAND 983040 (R 4520 8 ST)) -7) (ASH (MOD (ASH
    (R 4520 8 ST) -15) 2) 8)) -8) 24) (ASH (LOGIOR (ASH (MOD (ASH (R 4520 8 ST) -16) 2) 31) (ASH (LOGAND 61440 (R 4520 8 ST)) 15) (ASH
    (MOD (ASH (R 4520 8 ST) -11) 2) 26) (ASH (MOD (ASH (R 4520 8 ST) -12) 2) 25) (ASH (LOGAND 3840 (R 4520 8 ST)) 13) (ASH (MOD (ASH (R
    4520 8 ST) -7) 2) 20) (ASH (MOD (ASH (R 4520 8 ST) -8) 2) 19) (ASH (LOGAND 240 (R 4520 8 ST)) 11) (ASH (MOD (ASH (R 4520 8 ST) -3)
    2) 14) (ASH (MOD (ASH (R 4520 8 ST) -4) 2) 13) (ASH (MOD (R 4520 8 ST) 16) 9) (ASH (ASH (R 4520 8 ST) -31) 8)) -8)) (R (+ 4376 (* 8
    (R 4536 8 ST)) (* 8 (- (R 4528 8 ST)))) 8 ST)) -40) 256) -2) 2) 4))
```

While the first argument of the `LOGIOR` is easy to bound the second and third are problematic. `Ainni` bounds the `LOGIOR` to [0,63] in less than one hundredth of a second on a MacBook Pro laptop with a 2.6 GHz Intel Core i7 processor.

By the way, the second argument of the `LOGIOR` above actually lies in [0,15] and the third in [0,16]. But proving those two bounds with, say, `arithmetic-5/top`, takes about 33 seconds each, without `Ainni` and `meta-<`. But the main point is that `Ainni` *infers* a correct bound.

[6] We anticipate that any ISA employing Stateman's byte-addressed memory would use twos-complement arithmetic.

6 Syntactic Simplification of MOD Expressions

Machine arithmetic introduces many `MOD` expressions in which the second argument is constant. Stateman provides the extended metafunction `meta-MOD` that implements the following rules, where *i*, *j*, and *k* are natural constants. The function also uses a concept called "syntactic integer" realized by a function which takes the quotation of a term and determines whether it is obviously integer valued. For example, a sum expression is a syntactic integer provided the two arguments are syntactic integers, an `ASH` expression is a syntactic integer provided the first argument is, and a `LOGAND` expression is a syntactic integer regardless of the shape of the arguments. In the rules below, $x, x_1, \ldots, x_j$ must be syntactic integer expressions.

- `(MOD` x `0)` = x
- `(MOD` i k`)` can be computed if both arguments are constants
- `(MOD (MOD` z j`)` k`)` = `(MOD` z `j)`, if $j \leq k$
- `(MOD (MOD` x j`)` k`)` = `(MOD` x k`)`, if k divides j
- `(MOD (R` a i st`)` k`)` = `(R` a i st`)`, if $256^i \leq k$
- `(MOD (+` x_1 ... `(MOD` x j`)` ... x_j`)` k`)` = `(MOD (+` x_1 ... x ... x_j`)` k`)`, if k divides j
- `(MOD` x k`)` = x, if `Ainni` claims the upper bound of x is below k

The last rule is not only applied to the argument of `meta-MOD` but also to the output of the second-to-last rule.

Some of these rules are built into `arithmetic-5/top` but in the interests of speed, Stateman does not export `arithmetic-5/top` and does much arithmetic simplification in its metafunctions.

7 Some Details of Meta-R and Meta-!R

The most complicated of the metafunctions are `meta-R` and `meta-!R`, which use all of the functionality described above. The former is actually more complicated than the latter because the former deals with mixed read-over-write. We briefly discuss some design issues for these two functions, starting with the simpler, `meta-!R`, but we urge the interested reader to inspect the code in the Stateman book.

Since a successful application of `meta-!R` will transform `(!R` a `'`n v `(HIDE` st'`))` into `(HIDE (!R` a `'`n v st'`))`, we must be careful not to fire the metafunction too soon: none of the subterms will be rewritten again! Thus `meta-!R` checks whether a or v contain terms that might still be rewritten, e.g., embedded `IF`s, unexpanded `LAMBDA` applications, or read-over-writes that have not yet been resolved. If such subterms are found, the metafunction does not fire and `(!R` a `'`n v `(HIDE` st'`))` continues to be subject to rewriting.

If we decide to fire, we remove all `HIDE`S in a and v; remember they are probably arithmetic/logical expressions formed by the semantics of an instruction operating on data extracted from memory and thus (controversially) hidden. When we remove `HIDE`s we actually compute the depth of the deepest `HIDE` first and then copy only that far into the term so as to avoid re-copying a honsed term.

Then we dive through st' looking for a perfect shadow of a write to a of n bytes. This is actually a little more complicated than just looking for a deeper `(!R` a n ...`)` because the addresses may not be fully normalized. By using `Ainni` we can identify some non-identical addresses that are semantically equivalent in the current context. As we dive through st' looking for a shadowed assignment we also compute its depth, so we can come back and delete it without further interval analysis.

Moving on to meta-R, the main complication is mixed read-over-write. The question is, given (R *a* 'n (!R *b* 'k *v* *st*)), does part of the answer lie within *v* or not? Ainni can be used to handle many general overlap questions but we prefer not to use Ainni if simpler techniques apply. For example, if both *a* and *b* are constants we can just skip over this !R or extract the appropriate bytes from *v* (remember *n* and *k* are constants). But more generally, we ask whether *a* and *b* are offsets from some common address, e.g., *a* might be (+ 8 *sp*) and *b* might be (+ 16 *sp*) where *sp* is some expression denoting, say, the stack pointer. While neither address is constant we can still determine whether reading *n* bytes from *a* takes us into the region written, by doing arithmetic on the two constant offsets (8 and 16 in this example) and the constants *n* and *k*. When no common reference address can be found, we use Ainni. Space does not permit further description of mixed read-over-write and we urge the reader to see the Stateman code.

Furthermore, space does not permit discussion of the proof issues. But correctness, guards, and well-formedness guarantees are all proved. Probably the most interesting and difficult proofs concerned mixed read-over-write and the validity of removing a deeply buried perfectly shadowed write *without* being able to determine whether intervening writes also shadow it, i.e., how do you justify transforming

```
(!R a n v1
    (!R b k w
        (!R a n v2 st)))
```

to

```
(!R a n v1
    (!R b k w st))
```

without knowing the relations between *a*, *n*, *b* and *k*? The formalization of the general result we need is an inductively proved LOCAL lemma, named LEMMA3 in stateman22.lisp, establishing the correctness of a function that deletes a perfectly shadowed write at an arbitrary depth. LEMMA3 is used in the proof of META-!R-CORRECT.

8 Memoization

We have experimented with memoization of the metafunctions introduced by Stateman. Memoization is theoretically useful in code proofs because the same symbolic state might be produced on different paths through the code. In addition, the contents of the same addresses might be read multiple times from the same state. On the other hand, memoization imposes an overhead and is thus not always worthwhile.

Memoization hits most often if all of the arguments are honsed rather than consed. For example, if f is memoized and one has typed (f '(a . b)) at the top-level, then the value of f on that cons pair is stored in the hash table for f. But if one then types (f (cons 'a 'b)) the memoized answer is not found and f is recomputed. In Common Lisp terms, the argument must be EQ not EQUAL. All of the state expressions produced by our metafunctions are honsed and thus uniquely represented. But this alone will not make (memoize 'meta-R), for example, particularly useful.

First, memoization cannot be applied to an extended metafunction because one of the arguments is the ACL2 (live) state. So meta-R, which takes state as an argument (because it is a requirement of extended metafunctions) but which ignores state, is defined in terms of a wrapper, memoizable-meta-R which does not take state and which takes only the type-alist from the metafunction context, not the whole context.

Second, the term argument of `meta-R` is of the form `(R` *a* `'`*n* `(HIDE` *st*′`))` and typically came from simplifying some `R`-term in the model. The `(HIDE` *st*′`)` is honsed because it was produced by one of our metafunctions. But the rest of the term is not. So we `hons-copy` it before calling the wrapper. These `hons-copy`s are not as expensive as they may seem because the (very large) states and values extracted from them are already honsed.

Third, we must similarly `hons-copy` the type-alist.

Thus,

```
(defun meta-R (x mfc state)
  (declare (xargs :stobjs (state)
                  :guard (pseudo-termp x))
           (ignore state))
  (memoizable-meta-R (hons-copy x)
                     (hons-copy (mfc-type-alist mfc))))
```

Experiments have indicated that it is not worthwhile memoizing `meta-I`, `meta-S`, `meta-!I` or `meta-!S`: they are too simple. We have settled on:

```
(memoize 'memoizable-meta-r)
(memoize 'memoizable-meta-!r)
(memoize 'memoizable-meta-mod)
(memoize 'memoizable-meta-<)
```

While `Ainni` is an obvious candidate for memoization, the functions above include all of `Ainni`'s callers so it is not worthwhile.

Finally, when a metafunction fires – even a metafunction with a well-formedness guarantee – the output is put into *quote normal form* by which we mean all ACL2 primitives applied to constants are evaluated to constants. That is, `(CONS '1 '2)` is not in quote normal form, but `'(1 . 2)` is. This reduction to quote normal form is done by applying the empty substitution to the term with the ACL2 utility `sublis-var1`. We have found it worthwhile to memoize this function, but only when the substitution is empty and the form being normalized is hidden (and thus probably one produced by our metafunctions and thus honsed).

```
(memoize 'sublis-var1
         :condition '(and (null alist)
                          (consp form)
                          (eq (car form) 'HIDE)))
```

(**Future Work**) More experimentation must be done before we are comfortable with these decisions. In addition, it might be practical to make well-formedness guarantees ensure quote normal form.

9 Preliminary Performance Results

We have tested Stateman on only one very stressful example. Roughly put the setup for this example (which is not provided here) is as follows: Using the state provided by Stateman, we defined an ISA for a register machine that provides conventional but realistic arithmetic/logical functionality, addressing

modes, and control flow. We then implemented a compiler from a subset of ACL2 into this ISA. After allocating declared arrays, constants, etc., the compiler uses the rest of the memory to provide a call stack whose stack and frame pointers are among the earlier addresses. The compiler then compiles a system of ACL2 functions and a main program as though it were running on a stack machine, e.g., (LOGAND *x y*) is compiled by compiling *x* and *y* so as to leave two items on the stack, and then laying down a block of code to pop those two items into temporary registers, apply the LOGAND instruction to those registers, and push the result. Addressing modes are used whenever possible to minimize the number of instructions needed. We then compiled an ACL2 implementation of the DES algorithm.[7] The result is a code block of 15,361 instructions. We then ran an experimental version of Codewalker on this code.

Using Codewalker and the state management techniques described here, ACL2 explores the code above and generates both clock and semantic functions for DES.[8]

The largest symbolic state in the decompilation of the DES algorithm represents one path through the 5,280 instructions in the decryption loop. The state contains 2,158,895 function calls consisting of one call of !I and !S each and 58 calls of !R to distinct locations. That state expression also contains 459,848 calls of R and 1,698,987 calls of arithmetic/logical functions such as +, and *, LOGAND, LOGIOR, LOGXOR, ASH, and MOD. The values written are often very large. The largest value expression written is given by a term involving 147,233 function applications, 31,361 of which are calls of R and the rest are calls of arithmetic/logical functions.

We would like to be able to compare the performance of the current version of Stateman to older techniques (in which rewrite rules alone are used to canonicalize symbolic states) but Codewalker is unable to complete the exploration of our implementation of DES using those older techniques. The time it takes to symbolically execute successive instructions increases alarmingly, sometimes apparently exponentially (depending on the instruction being executed) as the state sizes increase. Of course, one might address that with better rewrite rules, metafunctions, etc., but that was the origin of the Stateman project.

However, we can provide some timing statistics on different versions of Stateman. The times shown are times taken to generate the clock and semantics functions of our DES implementation on a MacBook Pro laptop with a 2.6 GHz Intel Core i7 processor with 16 GB of 1600 MHZ DDR3 memory. Times are as measured by time$ and reported as "realtime" on a otherwise unloaded machine.

Roughly put, guard verification saved 33 seconds, well-formedness guarantees saved 337 more seconds, honsing as opposed to consing the metafunction answers saved 124 more seconds even though no memoization was employed, and memoizing then saved 119 more seconds. Of particular interest is that well-formedness guarantees were an order of magnitude more effective than guard verification and that

[7] Warren Hunt provided the definitions of the ISA and the DES algorithm in ACL2.

[8] As of this writing the Codewalker exploration of DES does not perform its standard "projection" (the transformation of functions that describe the entire state to functions that describe the contents of specific state components) because ACL2 gets a stack overflow trying to handle states of such large size. (**Future Work**) Clearly, additional work is necessary on Codewalker and/or ACL2 itself to handle the terms being produced by Stateman.

honsing even without memoization was a win (presumably because less time was spent in allocation).

without guard verification, well-formedness guarantees, honsing or memoization	988 seconds
with guard verification but without well-formedness guarantees, honsing, or memoization	955 seconds
with guard verification and well-formedness guarantees, but without honsing or memoization	618 seconds
with guard verification, well-formedness guarantees, and honsing, but without memoization	494 seconds
with guard verification, well-formedness guarantees, honsing, and the memoization described	375 seconds

10 Acknowledgments

I especially thank Warren Hunt for his invaluable help during the development of this software. Warren developed the definitions and proved many of the basic rewrite rules for `I`, `S`, `R`, `!I`, `!S`, and `!R`, as well as an ACL2 implementation of DES and the formal semantics for the ISA to which the stack machine compiles. I thank Matt Kaufmann, who gave me some strategic advice on lemma development to prove the correctness of one of the metafunctions as well as his usual extraordinary efforts to maintain ACL2 while I pursue topics such as this one. Finally, the reviewers of this paper improved it significantly and I am grateful for their careful and constructive criticism.

References

[1] S. Goel, W.A. Hunt & M. 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, http://www.cs.utexas.edu/users/hunt/FMCAD/FMCAD14/proceedings/final.pdf, EPFL, Switzerland, pp. 91–98, doi:10.1109/FMCAD.2014.6987600.

[2] M. Kaufmann, P. Manolios & J S. Moore, editors (2000): *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Press, Boston, MA.

[3] M. Kaufmann, P. Manolios & J S. Moore (2000): *Computer-Aided Reasoning: An Approach*. Kluwer Academic Press, Boston, MA., doi:10.1007/978-1-4615-4449-4.

[4] M. Kaufmann & J S. Moore (2014): *The ACL2 Home Page*. In: *http://www.cs.utexas.edu/users/moore/acl2/*, Dept. of Computer Sciences, University of Texas at Austin.

[5] M. Kaufmann & J S. Moore (2015): *Well-Formedness Guarantees for ACL2 Metafunctions and Clause Processors*. In: *(submitted for publication).*

[6] H. Liu & J S. Moore (2004): *Java Program Verification via a JVM Deep Embedding in ACL2*. In K. Slind, A. Bunker & G. Gopalakrishnan, editors: *17th International Conference on Theorem Proving in Higher*

Order Logics: TPHOLs 2004, Lecture Notes in Computer Science 3223, Springer, pp. 184–200, doi:10.1007/978-3-540-30142-4_14.

[7] J S. Moore & M. Martinez (2009): *A Mechanically Checked Proof of the Correctness of the Boyer-Moore Fast String Searching Algorithm*. In: *Engineering Methods and Tools for Software Safety and Security (Proceedings of the Martoberdorf Summer School, 2008)*, IOS Press, pp. 267–284, doi:10.3233/978-1-58603-976-9-267.

[8] Magnus O. Myreen (2009): *Formal verification of machine-code programs*. Ph.D. thesis, University of Cambridge.

[9] Magnus O. Myreen, Konrad Slind & Michael J. C. Gordon (2012): *Decompilation into Logic Improved*. In: *Formal Methods in Computer-Aided Design (FMCAD), 2012*, pp. 78–81.

[10] E. Toibazarov (2013): *An ACL2 Proof of the Correctness of the Preprocessing for a Variant of the Boyer-Moore Fast String Searching Algorithm*. Honors Thesis, Computer Science Dept., University of Texas at Austin. See http://www.cs.utexas.edu/users/moore/publications/toibazarov-thesis.pdf.

Proving Skipping Refinement with ACL2s

Mitesh Jain Panagiotis Manolios
Northeastern University
{jmitesh,pete}@ccs.neu.edu *

We describe three case studies illustrating the use of ACL2s to prove the correctness of optimized reactive systems using skipping refinement. Reasoning about reactive systems using refinement involves defining an abstract, high-level *specification* system and a concrete, low-level *implementation* system. Next, one shows that the behaviors of the implementation system are allowed by the specification system. Skipping refinement allows us to reason about implementation systems that can "skip" specification states due to optimizations that allow the implementation system to take several specification steps at once. Skipping refinement also allows implementation systems to *stutter*, *i.e.*, to take several steps before completing a specification step. We show how ACL2s can be used to prove skipping refinement theorems by modeling and proving the correctness of three systems: a JVM-inspired stack machine, a simple memory controller, and a scalar to vector compiler transformation.

1 Introduction

Refinement is a powerful method for reasoning about reactive systems. The idea is that a simple high-level abstract system acts as a specification for a low-level implementation of a concrete system. The goal is then to prove that all observable behaviors of the concrete system are behaviors of the abstract system. It is often the case that the concrete system requires several steps to match one high-level step of the abstract system, a phenomenon commonly known as stuttering. Therefore, notions of refinement usually directly account for stuttering [2, 5, 8]. However, in the course of engineering an efficient implementation, it is often the case that a single step of the concrete system can correspond to several steps of the abstract system, a phenomenon that is dual of stuttering. For example, in order to reduce memory latency and effectively utilize memory bandwidth, memory controllers often buffer requests to memory. The pending requests in the buffer are analyzed for address locality and then at some time in the future, multiple locations in the memory are read and updated simultaneously. Similarly, to improve instruction throughput, superscalar processors fetch multiple instructions in a single clock cycle. These instructions are analyzed for instruction-level parallelism (*e.g.*, the absence of data dependencies), and where possible multiple instructions are executed in parallel, retired in a single clock cycle. In both the above examples, updating multiple locations in memory and retiring multiple instructions in a single clock cycle, results in scenario where a single step in the optimized implementation may correspond to multiple steps in the abstract system. A notion of refinement that only account for stuttering is therefore not appropriate for reasoning about such optimized systems.

In our companion paper [10], we proposed *skipping refinement*, a new notion of correctness for reasoning about optimized reactive systems and a proof method that is amenable for mechanical reasoning. The applicability of skipping refinement was shown using three case studies: a JVM-inspired stack machine, an optimized memory controller, and a vectorizing compiler transformation. In [10] we focused on finite-state models for the systems in the first two case studies and used model-checkers to verify

*This research was supported in part by DARPA under AFRL Cooperative Agreement No. FA8750-10-2-0233, by NSF grants CCF-1117184 and CCF-1319580, and by OSD under contract FA8750-14-C-0024.

M. Kaufmann and D. Rager (Eds.): ACL2 Workshop 2015 (ACL2 2015).
EPTCS 192, 2015, pp. 111–127, doi:10.4204/EPTCS.192.9

skipping refinement. In this paper, we consider their corresponding infinite-state models and prove their correctness in ACL2s, an interactive theorem prover [7]. We also discuss in detail the modeling of vectorizing compiler transformation and its proof of correctness. In Section 2, we motivate the need for a new notion of refinement with an example. In Section 3, we define well-founded skipping simulation. In Section 4 we discuss the three case studies. We end the paper with conclusion and future work in Section 5.

2 Motivating Example

To illustrate the notion of skipping simulation, we consider an example of a discrete-time event simulation (DES) system [10]. An abstract high-level specification of DES is described as follows. Let E be set of events and V be set of variables. Then a state of abstract DES is a three-tuple $\langle t, Sch, A\rangle$, where t is a natural number denoting current time; Sch is a set of pairs (e, t_e), where e is an event scheduled to be executed at time $t_e \geq t$; A is an assignment to variables in V. The transition relation for the abstract DES system is defined as follows. If at time t there is no $(e,t) \in Sch$, *i.e.*, there is no event scheduled to be executed at time t, then t is incremented by 1. Else, we (nondeterministically) choose and execute an event of the form $(e,t) \in Sch$. The execution of event may result in modifying A and also adding finite number of new pairs (e', t') in Sch. We require that $t' > t$. Finally execution involves removing the executed event (e,t) from Sch.

Now, consider an optimized, concrete implementation of the abstract DES system. As before, a state of the concrete system is a three-tuple $\langle t, Sch, A\rangle$. However, unlike the abstract system which just increments time by 1 when no events are scheduled for the current time, the optimized system uses a priority queue to find the next event to execute. The transition relation is defined as follows. An event (e, t_e) with the minimum time is selected, t is updated to t_e and the event e is executed, as in the abstract DES.

Notice that when no events are scheduled for execution at the current time, the optimized implementation of the discrete-time event simulation system can run faster than the abstract specification system by *skipping* over abstract states. This is not a stuttering step as it results in an observable change in the state of the concrete DES system (t is update to t_e). Also, it does not correspond to a single step of the specification. Therefore, it is not possible to prove that the implementation *refines* the specification using notions of refinement that only allow stuttering [2, 13], because that just is not true. But, intuitively, there is a sense in which the optimized DES system *does* refine the abstract DES system. The notion of skipping refinement proposed in [10] is an appropriate notion to relate such systems: a low-level implementation that can run slower (stutter) or run faster (skip) than the high-level specification.

3 Skipping Refinement

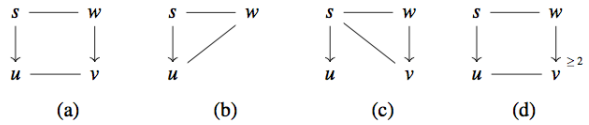
In this section, we first present the notion of well-founded skipping simulation [10]. The notion is defined in the general setting of labeled transition systems (TS) where labeling is on states. [1] We also place no restriction on the state space sizes and the branching factor, and both can be of arbitrary infinite cardinalities. The generality of TS is useful to model systems that may exhibit unbounded nondeterminism, for example, modeling a program in a language with random assignment command $x =?$, which sets x to an arbitrary integer [3].

[1] Note that labeled transition system are also used in the literature to refer to transition systems where transitions (edges) are labeled. However, we prefer to work with TS where states are labeled.

We first describe the notational conventions used in the paper. Function application is sometimes denoted by an infix dot "." and is left-associative. For a binary relation R, we often use the infix notation xRy instead of $(x,y) \in R$. The composition of relation R with itself i times (for $0 < i \leq \omega$) is denoted R^i ($\omega = \mathbb{N}$ and is the first infinite ordinal). Given a relation R and $1 < k \leq \omega$, $R^{<k}$ denotes $\bigcup_{1 \leq i < k} R^i$ and $R^{\geq k}$ denotes $\bigcup_{\omega > i \geq k} R^i$. Instead of $R^{<\omega}$ we often write the more common R^+. $\uplus$ denotes the disjoint union operator. Quantified expressions are written as $\langle Qx\colon r\colon p\rangle$, where Q is the quantifier (*e.g.*, $\exists, \forall$), x is the bound variable, r is an expression that denotes the range of x (*true*, if omitted), and p is the body of the quantifier.

Definition 1 *A labeled transition system (TS) is a structure $\langle S, \rightarrow, L\rangle$, where S is a non-empty (possibly infinite) set of states, $\rightarrow \subseteq S \times S$ is a left-total transition relation (every state has a successor), and L is a labeling function: its domain is S and it tells us what is observable at a state.*

Skipping refinement is defined based on well-founded skipping simulation, a notion that is amenable for mechanical reasoning. This notion allows us to reason about skipping refinement by checking mostly local properties, *i.e.*, properties involving states and their successors. The intuition is, for any pair of states s, w, which are related and a state u such that $s \rightarrow u$, there are four cases to consider (Definition 3): (a) either we can match the move from s to u right away, *i.e.*, there is a v such that $w \rightarrow v$ and u is related to v, or (b) there is stuttering on the left, or (c) there is stuttering on the right, or (d) there is skipping on the left.



Definition 2 (Well-founded Skipping) *$B \subseteq S \times S$ is a well-founded skipping relation on a transition system $\mathscr{M} = \langle S, \rightarrow, L\rangle$ iff:*

(WFSK1) $\langle \forall s, w \in S\colon sBw\colon L.s = L.w\rangle$

(WFSK2) There exist functions, $rankt\colon S \times S \rightarrow W$, $rankl\colon S \times S \times S \rightarrow \omega$, such that $\langle W, \prec\rangle$ is well-founded and

$\langle \forall s,u,w \in S : s \rightarrow u \wedge sBw :$

(a) $\langle \exists v\colon w \rightarrow v\colon uBv\rangle \vee$

(b) $(uBw \wedge rankt(u,w) \prec rankt(s,w)) \vee$

(c) $\langle \exists v\colon w \rightarrow v\colon sBv \wedge rankl(v,s,u) < rankl(w,s,u)\rangle \vee$

(d) $\langle \exists v : w \rightarrow^{\geq 2} v\colon uBv\rangle\rangle$

In the above definition, conditions (WFSK2a) to (WFSK2c) require reasoning only about single step $\rightarrow$ of the transition system. But condition (WFSK2d) requires us to check that there exists a v such that v is *reachable* from w in two or more steps and uBv holds. Reasoning about reachability, in general, is not local. However, for the kinds of optimized systems we are interested in the number of abstract steps that a concrete step corresponds to is bounded by a constant—a bound determined early on in the design phase. For example, the maximum number of abstract steps that a concrete step of a superscalar processor can correspond to is the number of instruction that the designer decides to retire in a single cycle. This is

a constant that is decided early on in the design phase. Therefore, for such systems we can still reason using "local" methods. Furthermore, in the case this constant is a "small" number, condition (WFKS2d) can be checked by simply unrolling the transition relation of the concrete system, an observation that we exploit in our first two case studies. On the other hand, this simplification is not always possible. For example, in the optimized DES system describe above, notice that number of abstract steps that optimized DES can take corresponds to the difference between current time and earliest time an event is scheduled for execution. This difference can not be a priori bounded by a constant.

We now define the notion of skipping refinement, a notion that relates *two* transition systems: an *abstract* transition system and a *concrete* transition system. In order to define skipping refinement, we make use of *refinement maps*, functions that map states of the concrete system to states of the abstract system. Informally, if the concrete system is a skipping refinement of the abstract system then its observable behaviors are also behavior of the abstract system modulo skipping (which includes stuttering). For example, in our running example of DES, if the refinement map is the identity function then it is easy to see that any behavior of the optimized system is a behavior of the abstract system modulo skipping. In practice, the abstract system and the concrete system are described at different levels of abstraction. Refinement maps along with the labeling function enable us to define what is observable at concrete states from the view point of the abstract system.

Definition 3 (Skipping Refinement) *Let $\mathscr{M}_A = \langle S_A, \xrightarrow{A}, L_A \rangle$ and $\mathscr{M}_C = \langle S_C, \xrightarrow{C}, L_C \rangle$ be transition systems and let $r\colon S_C \to S_A$ be a* refinement map. *We say $\mathscr{M}_C$ is a skipping refinement of $\mathscr{M}_A$ with respect to r, written $\mathscr{M}_C \lesssim_r \mathscr{M}_A$, if there exists a relation $B \subseteq S_C \times S_A$ such that all of the following hold.*

1. $\langle \forall s \in S_C :: sBr.s \rangle$ and
2. *B is an WFSK on $\langle S_C \uplus S_A, \xrightarrow{C} \uplus \xrightarrow{A}, \mathscr{L} \rangle$ where $\mathscr{L}.s = L_A(s)$ for $s \in S_A$, and $\mathscr{L}.s = L_A(r.s)$ for $s \in S_C$.*

Well-founded skipping gives us a simple proof rule to determine if a concrete transition system $\mathscr{M}_C$ is a skipping refinement of an abstract transition system $\mathscr{M}_A$ with respect to a refinement map r. Given a refinement map $r : S_C \to S_A$ and relation $B \subseteq S_C \times S_A$, we check the following two conditions: (a) for all $s \in S_C$, $sBr.s$ and (b) if B is a WFSK on the disjoint union of $\mathscr{M}_C$ and $\mathscr{M}_A$. If (a) and (b) hold, $\mathscr{M}_C \lesssim_r \mathscr{M}_A$. For a more detailed exposition of skipping refinement we refer the reader to our companion paper [10].

Notice that we place no restrictions on refinement maps. When refinement is used in specific contexts it is often useful to place restrictions on what a refinement map can do, *e.g.*, we may require for every $s \in S_C$ that $L_A(r.s)$ is a projection of $L_C(s)$. The generality of refinement map is useful in all three case studies considered in this paper, where a simple refinement map that is a projection function would not have sufficed.

4 Case Studies

We consider three case studies. The first case study is a hardware implementation of a JVM-inspired stack machine with an instruction buffer. The second case study is a memory controller with an optimization that eliminates redundant writes to memory. The third case study is a compiler transformation that vectorizes a list of scalar instructions. For each case study we model the abstract system and the concrete system in ACL2s. We define an appropriate refinement map and prove that the implementation refines the specification using well-founded skipping simulation.

We first briefly list some conventions used to describe the syntax and the semantics of the systems. Adding element e to the beginning or end of a list (or an array) l is denoted by $e :: l$ and $l :: e$, respectively.

Each transition consists of a *state* :$condition_1$,...,$condition_n$ pair above a line, followed by the next state below the line. If a concrete state matches the state in a transition and satisfies each of the conditions, then the state can transition to the state below the line. We formalize the operational semantics of the machines by describing the effect of each instruction on the state of the machine. The proof scripts are publicly available [1].

4.1 JVM-inspired Stack Machine

In this case study, we verify a stack machine inspired by the Java Virtual Machine (JVM). Java processors have been proposed as an alternative to just-in-time compilers to improve the performance of Java programs. Java processors such as JME [9] fetch bytecodes from an instruction memory and store them in an instruction buffer. The bytecodes in the buffer are analyzed to perform instruction-level optimizations *e.g.*, instruction folding. In this case study, we verify BSTK, a simple hardware implementation of part of the JVM. BSTK is an incomplete and inaccurate model of JVM that only models an instruction memory, an instruction buffer and a stack. Only a small subset of JVM instructions are supported (`push`, `pop`, `top`, `nop`). However, even such a simple model is sufficient to exhibit the applicability of skipping simulation and the limitations of current hardware model-checking tools.

STK is the high-level specification with respect to which we verify the correctness of BSTK, the implementation. Their behaviors are defined using abstract transition systems. The syntax and the operational semantics are shown in Fig. 1.

The state of STK consists of an instruction memory `imem`; a program counter `pc`; and a stack `stk`. An instruction is one of `push, pop, top`, and `nop`. We use the `listof` combinator in `defdata` to encode the instruction memory as list of instructions and stack as a list of elements [6]. The program counter is encoded as a natural number using the primitive data type `nat`. We then compound these components to encode state of STK using the `defdata record` construct. The `defdata` framework introduces a constructor function `sstate`, a set of accessor functions for each field (*e.g.*, `sstate-imem`), a recognizer function `sstatep` identifying the state, an enumerator `nth-sstate` and several useful theorems to reason about compositions of these functions.

```
(defdata el all)

(defdata stack (listof el))

(defdata inst (oneof (list 'pop) (list 'top)
                     (list 'nop) (list 'push el)))

(defdata inst-mem (listof inst))

(defdata sstate (record (imem . inst-mem)
                        (pc . nat)
                        (stk . stack)))
```

STK fetches an instruction from the instruction memory, executes it, increases the program counter, and possibly modifies the stack, as outlined in Fig. 1.

STK fetches an instruction from the `imem`, executes it, increments the `pc` by 1, and possibly modifies the `stk`, as outlined in Fig. 1. Since STK is a deterministic machine, we formalize its transition relation using a function `spec-step`, which uses an auxiliary function `stk-step-inst` to capture the effect of

$stk := [] \mid el :: stk$ (Stack)

$inst := \langle push\ e \rangle \mid \langle pop \rangle \mid \langle top \rangle \mid \langle nop \rangle$ (Instruction)

$imem := [] \mid inst :: imem$ (Program)

$pc := 0 \mid 1 \mid \cdots \mid n \mid \cdots$ (Program Counter)

$ibuf := [inst_1, \ldots, inst_k]$ (Instruction Buffer)

$sstate := \langle imem, pc, stk \rangle$ (STK State)

$istate := \langle imem, pc, ibuf, stk \rangle$ (BSTK State)

STK ($\xrightarrow{A}$) where s = capacity of stk, $t = |stk|$

$$\frac{\langle imem, pc, stk \rangle : imem[pc] = \langle push\ v \rangle, t < s}{\langle imem, pc+1, v :: stk \rangle}$$

$$\frac{\langle imem, pc, stk \rangle : imem[pc] = \langle push\ v \rangle, t = s}{\langle imem, pc+1, stk \rangle}$$

$$\frac{\langle imem, pc, [] \rangle : imem[pc] = \langle pop \rangle}{\langle imem, pc+1, [] \rangle}$$

$$\frac{\langle imem, pc, v :: stk \rangle : imem[pc] = \langle pop \rangle}{\langle imem, pc+1, stk \rangle}$$

$$\frac{\langle imem, pc, stk \rangle : imem[pc] = \langle top \rangle}{\langle imem, pc+1, stk \rangle}$$

$$\frac{\langle imem, pc, stk \rangle : imem[pc] = \langle nop \rangle}{\langle imem, pc+1, stk \rangle}$$

$$\frac{\langle imem, pc, stk \rangle : imem[pc] = nil}{\langle imem, pc+1, stk \rangle}$$

BSTK ($\xrightarrow{C}$) where k = capacity of $ibuf$, $m = |ibuf|$

$$\frac{\langle imem, pc, ibuf, stk \rangle : m < k,\ imem[pc] \neq \langle top \rangle}{\langle imem, pc+1, ibuf :: imem[pc], stk \rangle}$$

$$\frac{\langle imem, pc, ibuf, stk \rangle : imem[pc] = \langle top \rangle,\ \langle ibuf, 0, stk \rangle \xrightarrow{A}{}^{m} \langle ibuf, m, stk' \rangle}{\langle imem, pc+1, [], stk' \rangle}$$

$$\frac{\langle imem, pc, ibuf, stk \rangle : imem[pc] = nil,\ \langle ibuf, 0, stk \rangle \xrightarrow{A}{}^{m} \langle ibuf, m, stk' \rangle}{\langle imem, pc+1, [], stk' \rangle}$$

$$\frac{\langle imem, pc, ibuf, stk \rangle : m = k,\ \langle ibuf, 0, stk \rangle \xrightarrow{A}{}^{m} \langle ibuf, m, stk' \rangle}{\langle imem, pc+1, [imem[pc]], stk' \rangle}$$

Figure 1: Syntax and Semantics of Stack and Buffered Stack Machine

executing an instruction on the stack. We are now ready to define the transition system $\mathscr{M}_A$ of STK machine. The set of states S_A in the transition system $\mathscr{M}_A$ is is the set of all states satisfying the predicate `sstatep`. Two states $s, u \in S_A$ are related by transition relation $\xrightarrow{A}$ iff it is possible in one step to transition from s to u, *i.e.*, `u = (spec-step s)`. The labeling function, L_A is the identity function.

```
(defun stk-step-inst (inst stk)
 "returns next state of stk"
  (let ((op (car inst)))
    (cond ((equal op 'push)
           (mpush (cadr inst) stk ))
          ((equal op 'pop)
           (mpop stk))
          ((equal op 'top)
           (mtop stk))
          (t stk))))

(defun spec-step (s)
  "single step of STK machine"
  (let* ((pc (sstate-pc s))
         (imem (sstate-imem s))
         (inst (nth pc imem))
         (stk (sstate-stk s)))
    (if (instp inst)
        (sstate imem (1+ pc) (stk-step-inst inst stk))
      (sstate imem (1+ pc) stk))))
```

The state of BSTK is similar to STK, except that it also includes an instruction buffer `ibuf`. The instruction buffer is encoded as a list of instructions with an additional restriction on its capacity (`ibuf-capacity`). To encode `ibuf` in the `defdata` framework, we have at least two choices. We can use the `oneof defdata` construct to encode it as an empty list or list of one, two, or three instructions. Another way is to use the capability of the `defdata` framework to define custom data types. In the later case, we first define a recognizer function `inst-buffp` and an enumerator function `nth-inst-buff`.

```
(defun inst-buffp (l)
  (and (inst-memp l)
       (<= (len l) (ibuf-capacity))))

(defun nth-inst-buff (n)
  (let ((imem (nth-inst-mem n)))
    (if (<= (len imem) (ibuf-capacity))
        imem
      (let ((i1 (car imem))
            (i2 (cadr imem))
            (i3 (caddr imem)))
        (list i1 i2 i3)))))
```

We can now register our custom type `inst-buff` using the `register-custom-type` macro. Once we have registered it as a `defdata` type we can use it just like other type directly introduced using `defdata` construct.

```
(register-custom-type inst-buff :enumerator nth-inst-buff
                                :predicate inst-buffp)
```

We can now define state of BSTK machine using `defdata record` construct.

```
(defdata istate
  (record (imem . inst-mem)
          (pc . nat)
          (stk . stack)
          (ibuf . inst-buff)))
```

BSTK fetches an instruction from the instruction memory, and if the instruction fetched is not *top* and the instruction buffer is not full (function *stutterp* below), it queues the fetched instruction to the end of the instruction buffer and increments the program counter. If the instruction buffer is full, then the machine executes all buffered instructions in the order they were enqueued, thereby draining the instruction buffer and obtaining a new stack. It also updates the instruction buffer so that it only contains just the current fetched instruction. If none of the transition rules match, then BSTK drains the instruction buffer (if it is not empty) and updates the stack accordingly. Since BSTK is also a deterministic machine, we encode its transition relation ($\xrightarrow{C}$) as the function `impl-step`. Having defined the transition relation and the state of BSTK machine, we can define its transition system $\mathscr{M}_C$.

```
(defun stutterp (inst ibuf)
  "BSTK stutters if ibuf is not full or the current instruction is not 'top"
  (and (< (len ibuf) (ibuf-capacity))
       (not (equal (car inst) 'top))))

(defun impl-step (s)
  "single step of BSTK"
  (let* ((stk (istate-stk s))
         (ibuf (istate-ibuf s))
         (imem (istate-imem s))
         (pc (istate-pc s))
         (inst (nth pc imem)))
    (if (instp inst)
        (let ((nxt-pc (1+ pc))
              (nxt-stk (if (stutterp inst ibuf)
                           stk
                         (impl-observable-stk-step stk ibuf)))
              (nxt-ibuf (if (stutterp inst ibuf)
                            (impl-internal-ibuf-step inst ibuf)
                          (impl-observable-ibuf-step inst))))
          (istate imem nxt-pc nxt-stk nxt-ibuf))
      (let ((nxt-pc (1+ pc))
            (nxt-stk (impl-observable-stk-step stk ibuf))
            (nxt-ibuf nil))
        (istate imem nxt-pc nxt-stk nxt-ibuf)))))
```

Before we describe the correctness of BSTK based on skipping refinement, we first discuss why an existing notion of refinement such as stuttering refinement [12] will not suffice. If BSTK takes a step,

which requires it to drain its instruction buffer (the buffer is full or the current instruction fetched is `top`), then the stack will be updated to reflect the execution of all instructions in `ibuf`, something that is neither a stuttering step nor a single transition of the STK system. Therefore, it is not possible to prove that BSTK refines STK, using stuttering refinement and a refinement map that does not transform the `stack`.

We now formulate the correctness of BSTK based on the notion of skipping refinement. We show $\mathscr{M}_C \lesssim_r \mathscr{M}_A$, using Definition 2. We define the refinement map, but first we note that we do not have to consider all syntactically well-formed STK states. We only have to consider states whose instruction buffer is consistent with the contents of the instruction memory, so called *good states* [15]. One way of defining a good state is as follows: state s is good iff $pc \geq |ibuf|$ and stepping BSTK from $\langle imem, pc - |ibuf|, [], stk \rangle$ state for $|ibuf|$ steps yields state s, where $|ibuf|$ is number of instructions in the instruction buffer of state s. We define a predicate `good-statep` recognizing a good state and show that the set of good states is closed under the transition relation of BSTK.

```
(defun commited-state (s)
  (let* ((stk (istate-stk s))
         (imem (istate-imem s))
         (ibuf (istate-ibuf s))
         (pc (istate-pc s))
         (cpc (nfix (- pc (len ibuf)))))
    (istate imem cpc stk nil)))

(defun good-statep (s)
  "if state s is reachable from a commited-state in |ibuf| steps"
  (let ((pc (istate-pc s))
        (ibuf (istate-ibuf s)))
    (and (istatep s)
         (>= pc (len ibuf))
         (let* ((cms (commited-state s))
                (s-cms (cond ((endp ibuf)
                              cms)
                             ((endp (cdr ibuf))
                              (impl-step cms))
                             ((endp (cddr ibuf))
                              (impl-step (impl-step cms)))
                             ((endp (cdddr ibuf))
                              (impl-step (impl-step (impl-step cms))))
                             (t cms))))
           (equal s-cms s)))))

(defthm good-state-inductive
  (implies (good-statep s)
           (good-statep (impl-step s))))
```

The refinement map `ref-map`, a function from a set of good states to set of abstract states (`sstatep`) is defined as follows.

```
(defun ref-map (s)
  (let* ((stk (istate-stk s))
         (imem (istate-imem s))
         (pc (istate-pc s))
         (ibuf (istate-ibuf s))
         (ibuflen (len ibuf))
         (rpc (nfix (- pc ibuflen))))
    (sstate imem rpc stk)))
```

Given `ref-map`, we define B to be the binary relation induced by it, *i.e.*, sBw iff s is a good state and $w = \texttt{ref-map}(s)$.

Now observe that when the instruction is full or the current instruction is *top*, one step of BSTK corresponds to largest number of STK steps. In both cases, the BSTK machine executes all instructions in the instruction buffer and if the current instruction is *top*, it executes it as well. The condition WFSK2d in Definition 2 that requires us to reason about reachability, hence can easily be reduced to bounded reachability. Hence, we set $j = k+2$, where k is the capacity of the instruction buffer, and condition WFSK2d is $\langle \exists v : w \rightarrow^{<j} v : uBv \rangle$.

Since STK and BSTK are deterministic machines and STK does not stutter, we only need to define one rank function, a function from set of good states to non-negative integers.

```
(defun rank (s)
  "rank of an istate s is capacity of ibuf - |ibuf|"
   (- (ibuf-capacity) (len (istate-ibuf s))))
```

With above observations we simplify WFSK2 (Definition 2) to following condition.
For all `s,u` such that `s` and `u` are good states and `u = (ref-map s)`

$$\texttt{(ref-map s)} \xrightarrow{A}{}^{<k+2} \texttt{(ref-map u)} \vee$$
$$(\texttt{(ref-map u)} = \texttt{(ref-map s)} \wedge \texttt{(rank u)} < \texttt{(rank s)}) \quad (1)$$

Notice that since BSTK is deterministic, u is a function of s, so we can remove u from the above formula. Since $k+2$ is a constant, we can expand out $\xrightarrow{A}{}^{<k+2}$ using only $\xrightarrow{A}$ instead. We formalize Equation 1 in ACL2s by first defining a function `spec-step-skip-rel`, which takes as input STK states v and w and returns true only if v is reachable from w in `(ibuf-capacity)` + 1 steps.

```
(defthm bstk-skip-refines-stk
  (implies (and (good-statep s)
                (equal w (ref-map s))
                (equal u (impl-step s))
                (not (and (equal w (ref-map u))
                          (< (rank u) (rank s)))))
           (spec-step-skip-rel w (ref-map u))))
```

Once the definitions were in place, proving `bstk-skip-refines-stk` with ACL2s was straightforward. Next, we evaluated how amenable is SKS for automated reasoning, *i.e.*, using *only* symbolic simulation and no additional lemmas. We model BSTK with instruction buffer capacity of 2, 3, and 4. while no other restrictions were placed on the machines. In particular, the instruction memory (`imem`) and the stack (`stk`) component of the state for BSTK and STK machines are unbounded. The experiments were run on a 2.2 GHz Intel Core i7 with 16 GB of memory. For the BSTK with instruction buffer

capacity of 2 instructions, it took $\sim$ 12 minutes to complete the proof and for a BSTK with instruction buffer capacity of 3 instructions, it took $\sim$ 2 hours. For BSTK with instruction buffer capacity of 4 instructions the proof did not finish in over 3 hours.

4.2 Memory Controller

A memory controller is an interface between a CPU and a memory, and synchronizes communication between them. Designers implement several optimizations in a memory controller to maximize available memory bandwidth utilization and reduce the latency of memory accesses, known bottlenecks in optimal performance of programs. In this case study, we consider OptMEMC, a simple model of such an optimized memory controller. In our simplified model, a CPU is modeled as a list of memory request (*reqs*) and memory as a list of natural numbers (*mem*).

OptMEMC fetches a memory request from location *pt* in a queue of CPU requests, *reqs*. It enqueues the fetched request in the request buffer, *rbuf* and increments *pt* to point to the next CPU request in *reqs*. The capacity of *rbuf* is k, a fixed positive integer. If the fetched request is a *read* or the request buffer is full, then before enqueuing the request into *rbuf*, OptMEMC first analyzes the request buffer for consecutive write requests to the same address in the memory (*mem*). If such a pair of writes exists in the buffer, it marks the older write requests in the request buffer as redundant. Then it executes all the requests in the request buffer except the one that are marked redundant. Requests in the buffer are executed in the order they were enqueued. In addition to read and write commands, the memory controller periodically issues a *refresh* command to preserve data in memory. A *refresh* command reads all memory locations and immediately writes them back without modification. Refresh commands are required to periodically reinforce the charge in the capacitive storage cells in a DRAM. In effect, a refresh command leaves the data memory unchanged. We define the function `mrefresh` and prove that the memory is same before and after execution of the `refresh` command. This is the only property of `mrefresh` that we would require.

```
(defthm mrefresh-mem-unchanged
  (equal (mrefresh mem)
         mem))
```

To reason about the correctness of OptMEMC using skipping refinement, we define a high-level abstract system, MEMC, that acts as the specification for OPTMEMC. It fetches a memory request from the CPU and immediately executes the request. The syntax and the semantics of MEMC and OPTMEMC are given in Fig. 2, using the same conventions as described previously in the stack machine section.

We now formulate the correctness of OptMEMC based on the notion of skipping refinement. Let $\mathscr{M}_{\mathrm{A}} = \langle S_A, \xrightarrow{A}, L_A \rangle$ and $\mathscr{M}_{\mathrm{C}} = \langle S_C, \xrightarrow{C}, L_C \rangle$ be transition systems for MEMC and OptMEMC respectively. Like in the previous case study, we encode the state of the machines using `defdata` and formalize the transition relation of OptMEMC and MEMC using a step function that describes the effect of each instruction on the state of the machine. The labeling function L_A and L_C are the identity functions. Given a refinement map `ref-map` : $S_C \to S_A$, we use Definition 2 to show that $\mathscr{M}_C \lesssim_r \mathscr{M}_A$. As was the case with the previous case study, OptMEMC and MEMC are deterministic machines and MEMC does not stutter. WFSK2 (Definition 2) can again be simplified to Formula 1.

Once the definitions of the transition systems for the two machines were in place, it was straightforward to prove skipping refinement with ACL2s. Like in the previous case study, we also prove the theorem using *only* symbolic execution and no additional lemmas, for configurations of OPTMEMC with buffer capacity of 2 and 3. For OptMEMC with buffer capacity of 2, the final theorem was proved

$mem := [] \mid v :: mem$ (Memory)

$req := \langle write\ addr\ v \rangle \mid \langle read\ addr \rangle \mid \langle refresh \rangle$ (Request)

$pt := 0 \mid 1 \mid \cdots \mid n \mid \cdots$ (Request Location)

$reqs := [] \mid req :: reqs$ (Requests)

$rbuf := [req_1, \ldots, req_k]$ (Request Buffer)

$sstate := \langle reqs, pt, mem \rangle$ (MEMC State)

$istate := \langle reqs, pt, rbuf, mem \rangle$ (OptMEMC State)

MEMC ($\xrightarrow{A}$)

$$\frac{\langle reqs, pt, mem \rangle,\ reqs[pt] = \langle write\ addr\ v \rangle}{\langle reqs, pt+1, mem[addr] \leftarrow v \rangle}$$

$$\frac{\langle reqs, pt, mem \rangle,\ reqs[pt] = \langle read\ addr \rangle}{\langle reqs, pt+1, mem \rangle}$$

$$\frac{\langle reqs, pt, mem \rangle,\ reqs[pt] = \langle refresh \rangle}{\langle reqs, pt+1, mem \rangle}$$

OptMEMC ($\xrightarrow{C}$)
Let $|rbuf| = j$

$$\frac{\langle reqs, pt, rbuf, mem \rangle, \quad j < k, req \neq top}{\langle reqs, pt, rbuf :: reqs[pt], mem \rangle}$$

$$\frac{\begin{array}{c}\langle reqs, pt, rbuf, mem \rangle, \quad reqs[pt] = \langle read\ addr \rangle, \\ \langle rbuf, 0, mem \rangle \xrightarrow{A}{}^{j} \langle rbuf, j, mem' \rangle\end{array}}{\langle reqs, pt, [], mem' \rangle}$$

$$\frac{\begin{array}{c}\langle reqs, pt, rbuf, mem \rangle, \quad j = k, \\ \langle rbuf, 0, mem \rangle \xrightarrow{A}{}^{j} \langle rbuf, k, mem' \rangle\end{array}}{\langle reqs, pt, rbuf :: reqs[pt], mem' \rangle}$$

Figure 2: Syntax and Semantics of MEMC and OptMEMC

in ~ 2 minutes and with OptMEMC buffer capacity of 3, it took ~ 1 hour to prove the final theorem. The proof with buffer capacity of 4 instructions did not finish in over 3 hours.

4.3 Superword Level Parallelism with SIMD instructions

An effective way to improve the performance of multimedia programs running on modern commodity architectures is to exploit Single-Instruction Multiple-Data (SIMD) instructions (*e.g.*, the SSE/AVX instructions in x86 microprocessors). Compilers analyze programs for *superword level parallelism* and when possible replace multiple scalar instructions with a compact SIMD instruction that concurrently operates on multiple data [11]. In this case study, we illustrate the applicability of skipping refinement to verify the correctness of such a compiler transformation.

For the purpose of this case study, we make some simplifying assumptions: the state of the source and target programs (modeled as transition systems) is a three-tuple consisting of a sequence of instructions, a program counter and a store. We also assume that a SIMD instruction simultaneously operates on *two* sets of data operands and that the transformation analyzes the program at a basic block level. Therefore, we do not model any control flow instruction. Fig. 3 shows how two add and two multiply scalar instructions are transformed into corresponding SIMD instructions. Notice that the transformation does *not* reorder instructions in the source program.

$$\begin{array}{lllll} a & = & b & + & c \\ d & = & e & + & f \end{array} \rightarrow \begin{bmatrix} a \\ d \end{bmatrix} = \begin{bmatrix} b \\ e \end{bmatrix} +_{SIMD} \begin{bmatrix} c \\ f \end{bmatrix}$$

$$\begin{array}{lllll} u & = & v & \times & w \\ x & = & y & \times & z \end{array} \rightarrow \begin{bmatrix} u \\ x \end{bmatrix} = \begin{bmatrix} v \\ y \end{bmatrix} \times_{SIMD} \begin{bmatrix} w \\ z \end{bmatrix}$$

Figure 3: Superword Parallelism

The syntax and operational semantics of the scalar and vector machines are given in Fig. 4, using the same conventions as described previously in the stack machine section. We denote that $x,\ldots,y$ are variables with values $v_x,\ldots,v_y$ in *store* by $\{\langle x,v_x\rangle,\ldots,\langle y,v_y\rangle\} \subseteq store$. We use $[\![(sop\ v_x\ v_y)]\!]$ to denote the result of a scalar operation *sop* and $[\![(vop\ \langle v_a\ v_b\rangle\langle v_d\ v_e\rangle)]\!]$ to denote the result of a vector operation *vop*. Finally, we use $store|_{x:=v_x,\ldots,y:=v_y}$ to denote that variables $x,\ldots,y$ are updated (or added) to *store* with values $v_x,\ldots,v_y$. Notice that the language of a source program consists of scalar instructions while the language of the target program consists of both scalar and vector instructions. As in the previous two case studies, we model the transition relation of a program (both source and target program) by modeling the effect of an instruction on the state of machines.

We use the translation validation approach to verify the correctness of the vectorizing compiler transformation [4], *i.e.*, we prove the equivalence between a source program and the generated vector program. As in the previous two case studies, the notion of stuttering simulation is too strong to relate a scalar program and the vector program produced by the vectorizing compiler, no matter what refinement map we use. To see this, note that the vector machine might run exactly twice as fast as the scalar machine and during each step the scalar machine might be modifying the memory. Since both machines do not stutter, in order to use stuttering refinement, the length of the vector machine run has to be equal to the run of the scalar machine.

Let $\mathscr{M}_{\mathrm{A}} = \langle S_A, \xrightarrow{A}, L_A\rangle$ and $\mathscr{M}_{\mathrm{C}} = \langle S_C, \xrightarrow{C}, L_C\rangle$ be transition systems of the scalar and vector machines, respectively corresponding to the source and target programs. The vector program is correct iff $\mathscr{M}_C$ refines $\mathscr{M}_{\mathrm{A}}$. We show $\mathscr{M}_C \lesssim_r \mathscr{M}_A$, using Definition 2. Determining j, an upper-bound on skipping

$loc := \{x, y, z, a, b, c, \ldots\}$ (Variables)

$sop := add \mid sub \mid mul \mid and \mid or \mid nop$ (Scalar Ops)

$vop := vadd \mid vsub \mid vmul \mid vand \mid vor \mid vnop$ (Vector Ops)

$sinst := sop\langle z\ x\ y\rangle$ (Scalar Inst)

$vinst := vop\langle c\ a\ b\rangle\langle f\ d\ e\rangle$ (Vector Inst)

$sprg := [] \mid sinst :: sprg$ (Scalar Program)

$vprg := [] \mid (sinst \mid vinst) :: vprg$ (Vector Program)

$store := [] \mid \langle x, v_x\rangle :: store$ (Registers)

Scalar Machine ($\xrightarrow{A}$)

$$\frac{\langle sprg, pc, store\rangle,\ \{\langle x, v_x\rangle, \langle y, v_y\rangle\} \subseteq store,\ sprg[pc] = sop\langle z\ x\ y\rangle,\ v_z = [\![(sop\ v_x\ v_y)]\!]}{\langle sprg, pc+1, store|_{z:=v_z}\rangle}$$

Vector Machine ($\xrightarrow{C}$)

$$\frac{\langle vprg, pc, store\rangle,\ \{\langle x, v_x\rangle, \langle y, v_y\rangle\} \subseteq store,\ sprg[pc] = sop\langle z\ x\ y\rangle,\ v_z = [\![(sop\ v_x\ v_y)]\!]}{\langle vprg, pc+1, store|_{z:=v_z}\rangle}$$

$$\frac{\langle vprg, pc, store\rangle,\ vprg[pc] = vop\langle c\ a\ b\rangle\langle f\ d\ e\rangle,\ \{\langle a, v_a\rangle, \langle b, v_b\rangle, \langle d, v_d\rangle, \langle e, v_e\rangle\} \subseteq store,\ \langle v_c, v_f\rangle = [\![(vop\ \langle v_a\ v_b\rangle\langle v_d\ v_e\rangle)]\!]}{\langle vprg, pc+1, store|_{c:=v_c, f:=v_f}\rangle}$$

Figure 4: Syntax and Semantics of Scalar and Vector Program

that reduces condition WFSK2d in Definition 2 to bounded reachability is simple because the vector machine can perform at most 2 steps of the scalar machine at once; therefore $j = 3$ suffices.

We next define the refinement map. Recall that refinement maps are used to define what is observable at concrete states from viewpoint of the abstract system. Let *sprg* be the source program and *vprg* be the compiled vector program. We first define a function `pcT` that takes as input the vector machine's program counter `pc` and a vector program `vprg` and returns the corresponding value of the scalar machine's program counter.

```
(defun num-scaler-inst (inst)
  (cond ((vecinstp inst)
         2)
        ((instp inst)
         1)
        (t 0)))

(defun pcT (pc vprg)
"maps values of the vector machine's pc to the corresponding values of
the scalar machine's pc"
  (let ((inst (nth pc vprg)))
```

```
   (cond ((or (not (integerp pc))
              (< pc 0))
          0)
         ((zp pc)
          (num-scaler-inst inst))
         (t (+ (num-scaler-inst inst) (pcT (1- pc) vprg))))))
```

We next define a function `scalarize-vprg` that takes as input a vector program `vprg`. It walks through the list of instructions in `vprg` and translates each instruction in one the following ways: if it is a vector instruction it *scalarizes* it into a list of corresponding scalar instructions, else if it is a scalar instruction it returns the list containing the instruction itself (function `scalarize` below). The result of `scalarize-vprg` is a scalar program. Notice that this function is significantly simpler than the compiler transformation procedure. This is because the complexity of a compiler transformation typically lies in its analysis phase, which determines if the transformation is even feasible, and not in the transformation phase itself.

```
(defun scalarize (inst)
"scalerize a vector instruction"
  (cond ((vecinstp inst)
         (let ((op (vecinst-op inst))
               (ra1 (car (vecinst-ra inst)))
               (ra2 (cdr (vecinst-ra inst)))
               (rb1 (car (vecinst-rb inst)))
               (rb2 (cdr (vecinst-rb inst)))
               (rc1 (car (vecinst-rc inst)))
               (rc2 (cdr (vecinst-rc inst))))
           (case op
             (vadd (list (inst 'add rc1 ra1 rb1)
                         (inst 'add rc2 ra2 rb2)))
             (vsub (list (inst 'sub rc1 ra1 rb1)
                         (inst 'sub rc2 ra2 rb2)))
             (vmul (list (inst 'mul rc1 ra1 rb1)
                         (inst 'mul rc2 ra2 rb2))))))
        ((instp inst) (list inst))
        (t nil)))

(defun scalarize-vprg-aux (pc vprg)
"scalerize the vector program from [0,pc]"
  (if (or (not (integerp pc))
          (< pc 0))
      nil
    (let ((inst (nth pc vprg)))
     (cond
      ((zp pc) ;=0
       (scalarize inst))
      (t
       (append (scalarize-vprg (1- pc) vprg) (scalarize inst)))))))
```

```
(defun scalarize-vprg (vprg)
 (scalarize-vprg-aux (len vprg) vprg))
```

The refinement map `ref-map`: $S_C \rightarrow S_A$, now can be defined as follows.

```
(defun ref-map (s)
  (let* ((store (vstate-store s))
         (vprg (vstate-vprg s))
         (isapc (pcT (1- (vstate-pc s)) vprg)))
    (sstate isapc store (scalarize-vprg (len vprg) vprg))))
```

Given `ref-map`, we define B to be the binary relation induced by the refinement map, *i.e.*, sBw iff $s \in S_C$ and $w =$ `(ref-map s)`. Notice that since the machines do not stutter, WFSK2 (Definition 2) can be simplified as follows. For all $s,u \in S_C$ such that $s \xrightarrow{C} u$:

$$\texttt{(ref-map s)} \xrightarrow{A}{}^{<3} \texttt{(ref-map u)} \tag{2}$$

Since the vector machine is deterministic, u is a function of s, so we can remove u from the above formula, if we wish. Also, we can expand out $\xrightarrow{A}{}^{<3}$ to obtain a formula using only $\xrightarrow{A}$ instead. We prove the appropriate lemmas to prove the final theorem: vector machine refines scalar machine.

```
(defthm vprg-skip-refines-sprg
  (implies (and (vstatep s)
                (equal w (ref-map s)))
           (spec-step-skip-rel w (ref-map (vec-step s)))))
```

where `vstatep` is the recognizer for a state of vector machine; `vec-step` is a transition function for vector machine; and `spec-step-skip-rel` is a function that takes as input two states of scalar machine and returns true if the second is reachable from the first in less than three steps.

Note that $pcT(pc, vprg)$ can also be determined using a history variable and would be a preferable strategy from verification efficiency perspective.

5 Conclusion and Future Work

In this paper, we used skipping refinement to prove the correctness of three optimized reactive systems in ACL2s. The concrete optimized systems can run "faster" than the corresponding abstract high-level specifications. Skipping refinement is an appropriate notion of correctness for reasoning about such optimized systems. Furthermore, well-founded skipping simulation gives "local" proof method that is amenable for automated reasoning. Stuttering simulation and bisimulation have been used widely to prove correctness of several interesting systems [14, 16, 17]. However, we have shown that these notions are too strong to analyze the class of optimized reactive systems studied in this paper. Skipping simulation is a weaker and more generally applicable notion than stuttering simulation. In particular, skipping simulation can be used to reason about superscalar processors, pipelined processors with multiple instructions completion, without modifying the specification (ISA), an open problem in [16]. We refer the reader to our companion paper [10] for a more detailed discussion on related work.

For future work, we would like to develop a methodology to increase proof automation for proving correctness of systems based on skipping refinement. In [10], we showed how model-checkers can be used to analyze correctness for finite-state systems. Similarly, we would like to use the GL framework [18], a verified framework for symbolic execution in ACL2, to further increase the efficiency and automation.

Acknowledgments

We would like to thank Harsh Raju Chamarthi for help on the proof of vectorizing compiler transformation.

References

[1] *Skipping Simulation Model*. Available at `http://www.ccs.neu.edu/home/jmitesh/sks`.

[2] Martín Abadi & Leslie Lamport: *The Existence of Refinement Mappings*. In: *Theoretical Computer Science, 1991*. Available at `http://dx.doi.org/10.1016/0304-3975(91)90224-P`.

[3] Krzysztof R Apt & Gordon D Plotkin: *Countable nondeterminism and random assignment*. Available at `http://dx.doi.org/10.1145/6490.6494`.

[4] Clark W. Barrett, Yi Fang, Benjamin Goldberg, Ying Hu, Amir Pnueli & Lenore D. Zuck: *TVOC: A Translation Validator for Optimizing Compilers*. In: *CAV, 2005*. Available at `http://dx.doi.org/10.1007/11513988_29`.

[5] Michael C. Browne, Edmund M. Clarke & Orna Grumberg: *Characterizing Finite Kripke Structures in Propositional Temporal Logic*. In: *Theoretical Computer Science, 1988*. Available at `http://dx.doi.org/10.1016/0304-3975(88)90098-9`.

[6] Harsh Raju Chamarthi, Peter C. Dillinger & Panagiotis Manolios: *Data Definitions in the ACL2 Sedan*. In: *ACL2 2014*. Available at `http://dx.doi.org/10.4204/EPTCS.152.3`.

[7] Harsh Raju Chamarthi, Peter C. Dillinger, Panagiotis Manolios & Daron Vroon: *The ACL2 Sedan Theorem Proving System*. In: *TACAS 2011*. Available at `http://dx.doi.org/10.1007/978-3-642-19835-9_27`.

[8] Rob J. van Glabbeek: *The Linear Time-Branching Time Spectrum (Extended Abstract)*. In: *CONCUR 1990*. Available at `http://dx.doi.org/10.1007/BFb0039066`.

[9] David S Hardin: *Real-time objects on the bare metal: an efficient hardware realization of the Java TM Virtual Machine*. In: *ISORC, 2001*, doi:10.1109/ISORC.2001.922817.

[10] Mitesh Jain & Panagiotis Manolios: *Skipping Refinement*. In: *CAV, 2015*. Available at `http://dx.doi.org/10.1007/978-3-319-21690-4_7`.

[11] Samuel Larsen & Saman P. Amarasinghe: *Exploiting superword level parallelism with multimedia instruction sets*. In: *PLDI, 2000*. Available at `http://doi.acm.org/10.1145/349299.349320`.

[12] P. Manolios: *Mechanical verification of reactive systems*. Ph.D. thesis, University of Texas.

[13] Panagiotis Manolios: *A Compositional Theory of Refinement for Branching Time*. In: *CHARME 2003*. Available at `http://dx.doi.org/10.1007/978-3-540-39724-3_28`.

[14] Panagiotis Manolios: *Correctness of Pipelined Machines*. In: *FMCAD, 2000*. Available at `http://dx.doi.org/10.1007/3-540-40922-X_11`.

[15] Panagiotis Manolios & Sudarshan K. Srinivasan: *A computationally effecient method based on commitment refinement maps for verifying pipelined machines*. In: *MEMOCODE, 2005*. Available at `http://dx.doi.org/10.1109/MEMCOD.2005.1487914`.

[16] Sandip Ray & Warren A. Hunt Jr.: *Deductive Verification of Pipelined Machines Using First-Order Quantification*. In: *CAV 2004*. Available at `http://dx.doi.org/10.1007/978-3-540-27813-9_3`.

[17] Sandip Ray & Rob Sumners: *Specification and Verification of Concurrent Programs Through Refinements*. In: *Journal of Automated Reasoning*. Available at `http://dx.doi.org/10.1007/s10817-012-9258-1`.

[18] Anna Slobodová, Jared Davis, Sol Swords & Warren A. Hunt Jr.: *A flexible formal verification framework for industrial scale validation*. In: *MEMOCODE, 2011*. Available at `http://dx.doi.org/10.1109/MEMCOD.2011.5970515`.