EPTCS 249

Proceedings of the

14th International Workshop on the ACL2 Theorem Prover and its Applications

Austin, Texas, USA, May 22-23, 2017

Edited by: Anna Slobodova and Warren Hunt Jr.

Published: 2nd May 2017
DOI: 10.4204/EPTCS.249
ISSN: 2075-2180
Open Publishing Association

Preface

This volume contains the proceedings of the Fourteenth International Workshop on the ACL2 Theorem Prover and Its Applications, ACL2 2017, a two-day workshop held in Austin, Texas, USA, on May 22–23, 2017. ACL2 workshops occur at approximately 18-month intervals, and they provide a 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. 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 theorem-prover family.

The proceedings of 2017 ACL2 Workshop include the seven technical papers and one extended abstract that were presented at the workshop. Each submission received two or three reviews. The workshop also included three invited talks: *Using Mechanized Mathematics in an Organization with a Simulation-Based Mentality*, by Glenn Henry of Centaur Technology, Inc.; *Formal Verification of Financial Algorithms, Progress and Prospects*, by Grant Passmore of Aesthetic Integration; and *Verifying Oracle's SPARC Processors with ACL2* by Greg Grohoski of Oracle. The workshop also included several rump sessions discussing ongoing research and 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. And we thank EPTCS and the arXiv for publishing the workshop proceedings in an open-access format. We also like to thank Centaur Technology and Oracle Corporation for their generous sponsorship of the workshop.

Warren A. Hunt, Jr. and Anna Slobodova
May, 2017

Warren Hunt Jr. and Anna Slobodova (Eds.):
ACL2 Theorem Prover and its Applications
EPTCS 249, 2017, pp. i–ii, doi:10.4204/EPTCS.249.0

ACL2 2017 Program Committee

Alessandro Coglio	Kestrel Institute
John Cowles	University of Wyoming
Mark Greenstreet	University of British Columbia
David Greve	Rockwell-Collins, Inc.
David Hardin	Rockwell-Collins, Inc.
Warren Hunt, Jr.	University of Texas
Matt Kaufmann	University of Texas
Pete Manolios	Northeastern University
J Moore	University of Texas
Rex Page	University of Oklahoma
Dmitry Nadezhin	Oracle Corporation
David Rager	Oracle Corporation
Sandip Ray	NXP Semiconductors
Jose-Luis Ruiz-Reina	University of Seville
David Russinoff	ARM, Inc.
Anna Slobodova	Centaur Technology, Inc.
Rob Sumners	Centaur Technology, Inc.
Freek Verbeek	Open University, The Netherlands

Additional Reviewers

Marc Schoolderman	Radbound University Nijmegen, The Netherlands

Table of Contents

The x86isa Books: Features, Usage, and Future Plans

Shilpi Goel
Department of Computer Science, University of Texas at Austin*
shigoel@cs.utexas.edu

The x86isa library, incorporated in the ACL2 community books project, provides a formal model of the x86 instruction-set architecture and supports reasoning about x86 machine-code programs. However, analyzing x86 programs can be daunting — even for those familiar with program verification, in part due to the complexity of the x86 ISA. Furthermore, the x86isa library is a large framework, and using and/or contributing to it may not seem straightforward. We present some typical ways of working with the x86isa library, and describe some of its salient features that can make the analysis of x86 machine-code programs less arduous. We also discuss some capabilities that are currently missing from these books — we hope that this will encourage the community to get involved in this project.

1 Introduction

The ACL2 community books contain several machine models (Y86, JVM, etc.) and libraries that aid in program verification (COI, Stateman, Codewalker, etc.). The x86isa library (:doc x86isa) adds to this repertoire by providing yet another formal, executable machine model — that of the single-processor x86 instruction-set architecture, with a specification of 400+ opcodes executing in Intel's 64-bit mode of operation. This library also offers the following capabilities:

- A tool to read, parse, and load an executable file (Mach-O [4] and ELF [6] formats) at the appropriate memory location of the x86 state;
- Utilities along the lines of the GNU Debugger (GDB) and Pintool [2] to monitor concrete program runs, and
- Books that provide rules that facilitate symbolic simulation of x86-64 machine-code programs.

Also, there are examples that illustrate how the above were used to set up the model for a program run, dynamically instrument a program, run co-simulations against an actual x86 processor for model validation, and perform x86 machine-code proofs. Consequently, the x86isa library, which is still in active development, is large — currently, it consists of around 60,000 lines of ACL2 (not counting automatically generated events) and around 240 files.

Just the complexity and size of the x86 ISA can deter people from being serious practitioners of x86 machine-code verification. Therefore, formal tools built to support this undertaking have an obligation to be easily accessible to the users — at least to those who already have some familiarity with program verification. To this end, we describe the x86isa library so that a user can find it relatively straightforward to get started with x86 machine-code analysis. We also present some important capabilities that are currently missing from this library. We hope that this will encourage the ACL2 community to contribute to this project, both by way of adding new features themselves and by way of providing feedback that will help x86isa become sophisticated over time.

*The author is now at Centaur Technology, Inc., but this work was done as a part of the author's PhD at UT Austin.

Warren Hunt Jr. and Anna Slobodova (Eds.):
ACL2 Theorem Prover and its Applications
EPTCS 249, 2017, pp. 1–17, doi:10.4204/EPTCS.249.1

2 Overview

The x86 ISA model has been developed using the classical *interpreter-style of operational semantics* that is prevalent in the ACL2 community: a recursively-defined interpreter over the x86 state is used to ascribe semantics to x86-64 programs. Models written using this style have the following main components: a machine state, semantic functions that describe instructions' behavior, a step function that executes one instruction by calling the appropriate instruction semantic function, and a run function (i.e., the interpreter) that calls the step function iteratively. We briefly describe our x86 ISA model in this section, referring an interested reader elsewhere [13, 10] for details.

2.1 x86 State

The x86 ISA state has been specified using an abstract stobj [12] called `x86`. The x86 ISA components currently supported by our model are: general-purpose registers (`rax`, `rbx`, etc.), instruction pointer, flags register, segment registers (`cs`, `ss`, etc.), memory-management registers (`gdtr`, `ldtr`), interrupt/task management registers (`idtr`, `tr`), control registers (`cr0`, `cr1`, etc.), floating-point registers (e.g., `fp-data0`, `mmx0`[1], `fp-ctrl`, `fp-status`, etc.), XMM registers, MXCSR register, machine-specific registers[2], and a byte-addressable main memory that specifies 2^{52} bytes. Additionally, `x86` contains some fields that control and report on the model's operation, rather than that of the machine. An example of such a field is the model state `ms` — if a model-related error occurs at any point during the execution of a program (e.g., an unimplemented opcode is encountered), then this field is populated with information about the error and execution is halted. Thus, the x86 ISA model is expected to reflect the real machine's state only if the `ms` field is empty. We discuss other such fields — the `user-level-mode`, `page-structure-marking-mode`, `undef`, `os-info`, and `env` — later in this paper.

2.2 Modes of Operation and x86 Memory Interface

Reasoning about *all* of the x86 machine code involved in the execution of a user-level (application) program is a huge undertaking. In addition to the x86 code corresponding to the application program itself, one would need to consider the x86 code corresponding to the underlying system programs as well. For example, consider a C program that uses `printf` to print "Hello, world!" to standard output — `printf` is a standard C library function that ultimately relies on the `write` system call provided by the OS. Statically compiling this program on an x86 platform generates an executable file of size 0.8MB![3]

For expediency during application program verification, a user may wish to assume, either temporarily or permanently, that the underlying system programs behave as expected. To this end, the x86 ISA model provides two main modes of operation: the *system-level mode* and the *user-level mode*. The x86 ISA model operates in the user-level mode when the `user-level-mode` field in `x86` is non-nil; otherwise, it operates in the system mode. Furthermore, the system-level mode offers two sub-modes of operation — the *marking* and *non-marking* mode; a non-nil value in the field `page-structure-marking-mode` dictates that the model operate in the system-level marking mode. These two sub-modes are used to optimize reasoning about certain kinds of system programs, and are discussed later in

[1] As dictated by the x86 ISA, MMX registers are aliased to the low 64 bits of the FPU's data registers.

[2] Intel defines many MSRs. Our model currently supports only 6 of them: `ia32_efer`, `ia32_fs_base`, `ia32_gs_base`, `ia32_kernel_gs_base`, `ia32_lstar`, `ia32_star`, `ia32_fmask`.

[3] This program was statically compiled on an Intel Xeon CPU (E31280) using the default options of GCC compiler, version 4.8.4. The standard C library used was Ubuntu EGLIBC, version 2.19.

Section 4.2. For now, we just note that the "true" specification of the x86 ISA is given by the model's system-level marking mode of operation, and any discussion about the system-level mode pertains to this sub-mode unless specified otherwise.

The system-level mode is intended to provide the same environment to a program as is provided by an x86 processor, and is suitable for the verification of OS routines. The user-level mode is intended for the verification of application programs under the assumption that the relevant OS services are correct. In this mode, the x86 system state — which includes some memory-resident data structures like the page tables — is hidden from the programs. The system-level and user-level modes share a large part of their code base, but they differ significantly in the view of their memory and the implementation of certain instructions — we discuss the latter in Section 2.3.

The x86 processors offer two main kinds of memories — *linear memory* and *physical memory* — which are indexed by linear and physical addresses, respectively. Physical memory is the RAM addressed by a processor on its bus. Linear memory is an abstraction of the physical memory that is offered to x86 programs via a memory management mechanism called *paging*. Paging is used to map a linear address to physical address using information present in ISA-specified, memory-resident data structures called the *page tables*. 64-bit programs cannot access physical memory directly; however, privileged 64-bit programs can alter the linear-to-physical address mapping by modifying the page tables.

The system-level mode of the x86 ISA model includes a specification of paging, and thus, it has a model of both linear and physical memory. In this mode, every linear memory access is translated to the corresponding physical memory access. The user-level mode has a model only of the linear memory because application programs typically do not get adequate privileges for directly interacting with the system data structures. The same memory field in `x86` is configured to specify physical memory in the system-level mode and linear memory in the user-level mode. To facilitate code sharing between these modes, we provide a uniform linear memory interface, where top-level memory accessor and updater functions call the appropriate mode-specific functions. This prevents us from needing to define two versions of an x86 ISA specification function.

Both the modes of operation specify yet another x86 memory management mechanism: *segmentation*. The system-level mode models segmentation in its full detail, whereas the user-level model captures only its application-level view. We omit details about this mechanism here because segmentation is mostly disabled in the 64-bit mode.

2.3 Instruction Semantic Functions

The behavior of each instruction can be defined in terms of reads from and writes to the x86 state. For example, an `add` instruction reads the source operands from the x86 state and then writes the following to the x86 state: the appropriately-sized sum, the updated flags, and the new value of the instruction pointer. Of course, this is a largely incomplete description of `add`; we have omitted important details — such as how operand sizes are determined, when exceptions are thrown, etc. — from this description.

We have modeled 413 x86 instruction opcodes, including arithmetic, floating-point, and control-flow instructions. The x86 ISA model also contains a specification of some system-mode instructions like `lgdt`, `lldt`, `lidt`, etc. — these instructions are available only in the system-level mode of operation. The list of instructions that are specified by our x86 ISA model can be found at `:doc x86isa::implemented-opcodes`.

2.3.1 Undefined and Random Values

An important part of defining instruction semantic functions is accounting for undefined and/or random behavior that is inherent in certain instructions. For example, many commonly-used instructions like `mul` and `div` leave certain flags undefined, and the `rdrand` instruction returns random values. We specify undefined and random values with the function `undef-read`, which simply invokes the function `undef-read-logic`.

```
(defun undef-read-logic (x86)
   ;; Declarations elided.
  (let* ((undef-seed (nfix (undef x86)))
         (new-unknown (create-undef undef-seed))
         (x86 (!undef (1+ undef-seed) x86)))
    (mv new-unknown x86)))

(defun-notinline undef-read (x86)
  ;; Declarations elided.
  (undef-read-logic x86))
```

The functions `undef` and `!undef` are the native accessor and updater functions of the `undef` field in the x86 state. The function `create-undef` is a constrained function, and its only known property is that it always returns a `natp`. After admitting `undef-read-logic`, we make `!undef` untouchable (see `:doc push-untouchable`) to ensure that `undef-read-logic` is the only function that can modify this `undef` field. Also, we never use `create-undef` in any function other than `undef-read-logic`.

The upshot of all of this arrangement is that `undef-read-logic` always returns an *indeterminate* value that can be used to specify either an undefined or a random value[4]. Every call of `undef-read-logic` produces a value that is equal to `create-undef` invoked with the current value of the `undef` field, and the `undef` field is incremented every time `undef-read-logic` is called. Since `!undef` and `create-undef` are never used outside `undef-read-logic`, `create-undef` always gets unique arguments. Essentially, this arrangement gives us a pool of indeterminate values that can be used when required.

We need to model all possible behaviors resulting from an indeterminate value while reasoning, but an appropriate concrete value is needed during execution. To this end, we use `undef-read` (instead of `undef-read-logic`) as our top-level specification function, and re-define it under the hood using `include-raw` so that `undef-read-logic` is used for reasoning and `undef-read-exec` is used for execution; we omit the definition of `undef-read-exec` here. Note that `undef-read` is directed to not be inlined by the Lisp compiler because re-definition of inlined functions may result in unpredictable behavior.

```
(defun undef-read$notinline (x86)
  ;; Declarations elided.
  (when
      ;; When the x86 model is being used for reasoning:
      (or (equal (f-get-global
                  'in-prove-read ACL2::*the-live-state*)
```

[4]Indeterminate values have the following useful property: the result of an equality test of an indeterminate value with any other value is unknown.

```
                t)
          (equal (f-get-global
                  'in-verify-read ACL2::*the-live-state*)
                 t))
    (return-from X86ISA::undef-read$notinline
      (X86ISA::undef-read-logic x86)))
  ;; When the x86 model is being used for concrete execution:
  (undef-read-exec x86))
```

2.3.2 System Calls and Non-Determinism

In addition to the memory model and availability of system-level instructions, a significant difference between the system-level and user-level modes is in their treatment of system calls. System calls are requests for services made by an application program to the OS. The `syscall` instruction, used by application programs, transfers control to a more privileged system sub-routine. The corresponding instruction `sysret` is used by system sub-routines to transfer control back to the application program. In the system-level mode, these two instructions are modeled as per their specifications in the Intel ISA manuals [1]. In the user-level mode, the specification of `syscall` has been extended to provide the semantics of some commonly used system calls like `read`, `write`, `open`, `close`, `lseek`, `dup`, `link`, and `unlink`. These system calls are OS-specific — for instance, the specification of `read` on FreeBSD is somewhat different from that on Linux. The `os-info` field in the x86 state is used to identify the OS under consideration so that the appropriate semantic function for these system calls can be chosen. The extended semantics of `syscall` is intended to capture system call behavior in its entirety, right from its invocation to its return, and therefore, the `SYSRET` instruction is unavailable in the user-level mode. We validate our system call specification functions by running co-simulations against the real machine plus the chosen OS.

System calls can exhibit different behaviors for different runs, even if given the same inputs — thus, they are non-deterministic from the point of view of an application programmer. Consider a `read` system call that is invoked to read from a file. It is possible that one run be successful, but another result in failure if the file has been deleted. In order to formally characterize the interaction of an application program with the underlying OS, we model an external environment using the `env` field in `x86` — this field represents the part of the external world that affects or is affected by system calls. The `env` field includes a file system and an oracle sub-field that specifies the result of non-deterministic computations. For example, the file descriptor (or handle) assigned to a file by the `open` system call is the lowest-numbered 32-bit file descriptor not currently open for that process — this descriptor may be different for different invocations of the system call, and thus, we obtain it from the oracle. The oracle is a map of linear addresses to a list of values; if it needs to be consulted during the execution of an instruction, then the first value in the list corresponding to the address of the instruction is returned. It is the user's responsibility to initialize `env`, and hence, the oracle, appropriately while reasoning — this provides a way to state precisely the expectations from the environment. For instance, when reasoning about the `open` system call, the `env` field in the initial x86 state can be constrained in such a way that the oracle returns an arbitrary 32-bit natural number that can be used as a file descriptor.

Our system call specification functions are re-defined in the same manner as `undef-read` so that foreign C/Assembly functions[5] are invoked during concrete executions — these foreign functions request

[5] We rely on CCL's Foreign Function Interface for the execution of system calls.

the system call service from the underlying OS (i.e., the host OS running ACL2) and return the results to the ACL2 caller function. The os-info field (and for that matter, env too) is irrelevant during concrete executions.

Note that the env field could have been used to specify undefined and random values too — the user could constrain the oracle to contain appropriate symbolic values (corresponding to the undefined and random values) at appropriate linear addresses (corresponding to the linear addresses of the instructions that generate these indeterminate values). Why then do we use the arrangement with the undef field? One reason is that these fields serve separate purposes. The env field is used to specify non-deterministic behavior resulting from reliance on an external environment, whereas the undef field is used to model indeterminateness in the ISA itself. Another reason is convenience — if env were used instead of undef, the user would have to initialize the oracle in env whenever instructions that write undefined or random values in some state components are to be executed. Such instructions — especially those that leave some flags undefined, like mul and div — are encountered frequently in a typical program, and using the undef field saves the user quite a lot of work.

2.4 Interpreter

The x86 step function, called x86-fetch-decode-execute, fetches the next instruction from the memory (which is located at the address in the instruction pointer register rip), decodes it, and then calls the appropriate instruction semantic function. The run function, x86-run, is the x86 ISA interpreter. Its definition is straightforward:

```
(defun x86-run (n x86)
  ;; Declarations elided.
  ;; Halt if there is a problem indicated by the ms
  ;; field, or if there are no more instructions
  ;; left to execute.
  (cond ((ms x86) x86)
        ((zp n) x86)
        (t (let* ((x86 (x86-fetch-decode-execute x86))
                  (n (1- n)))
             (x86-run n x86)))))
```

3 Dynamic Instrumentation of x86 Programs

Like most machine models written in ACL2, the x86 ISA model is also executable. The execution speed of the model is around 3.3 million instructions/second in the user-level mode and around 320,000 instructions/second in the system-level mode with a set-up of 1G page-table configuration[6]. One can validate the model against a real x86 processor by performing co-simulations. The model can be used as an instruction-set simulator to inspect the behavior of x86 machine-code programs by running concrete tests. It is generally a good idea to run such tests before reasoning about a program — testing may reveal "obvious" bugs quickly and may also help in program comprehension. We describe how to set up the x86 ISA model for a concrete run of a program and how to dynamically instrument a program run.

[6] This speed was measured on a Intel Xeon E31280 CPU @ 3.50GHz with 32GB RAM.

3.1 Initializing the x86 ISA Model for a Concrete Run

If not already available, obtain the x86 machine-code version of the program to be executed — for instance, a given C source program `foo.c` can be compiled on an x86 machine using GCC or LLVM to obtain the executable file `foo.o`. The file `foo.o` contains information that is necessary to execute the program, such as the x86 machine code itself, program's data, linear addresses where the various sections of the program must be placed, etc. Independently of the x86 ISA model, one can examine executable files using tools such as `objdump` and `otool`.

Now all we have to do is arrange for `foo.o` to be read in and parsed by the x86 model and then initialize the x86 state appropriately based on the information in `foo.o`. To this end, we recommend creating a fresh file, say `foo-run.lsp`, which contains the following events:

1. Include the top-level `x86isa` book.

   ```
   (in-package "X86ISA")
   (include-book "projects/x86isa/top" :dir :system :ttags :all)
   ```

2. The default value of `user-level-mode` field in the x86 state is `t`. Thus, if operation in the user-level mode is required (probably because `foo.c` is an application program), then go to the next step. However, in the system-level mode, the x86 state includes a model of the physical memory. Since `foo.o` contains memory locations in the linear address space, the paging data structures must be set up *before* `foo.o` is read and loaded into the x86 state.

   We provide a function `init-system-level-mode` that switches the model to the system-level mode by writing `nil` to the `user-level-mode` field and loads our default configuration of paging data structures in the model's physical memory at a specified address, say 0 for this contrived example. This value 0 is also written to the control register `cr3` so that the processor knows where to locate the paging data structures in the physical memory. Our default data structures simply provide an identity mapping from linear to physical addresses.

   ```
   (init-system-level-mode 0 x86)
   ```

   A user can choose to load his own configuration of paging data structures by writing to the physical memory in the x86 state directly.

3. The program in `foo.o` can be read and loaded into the memory of the x86 state by using `binary-file-load`. At this time, `binary-file-load` supports only ELF [6] (commonly used on Unix systems) or Mach-O [4] (commonly used on FreeBSD/Darwin systems) binaries.

   ```
   (binary-file-load "foo.o")
   ```

   Note that if instead of an executable file, the x86 machine-code program is available simply as a list of bytes intended to be located at a particular linear memory location (or some other such formulation), it is straightforward to load it into the model by simply writing these bytes to the memory — see the following step.

4. Other components of the x86 state, like the instruction pointer, registers, etc., can be initialized by using `init-x86-state`.

   ```
   (init-x86-state
     <initial contents of MS field --- typically nil>
     <initial value of the instruction pointer>
     <linear address where execution should halt>
     <initial values of various registers...>
   ```

```
 <updates to the memory>
x86)
```

Alternatively, one may use the stobj's native updater functions to write to the x86 state.

5. Run the program by executing `x86-run` — the run function will either execute <*n*> number of steps, or terminate early if either an error is encountered or if the instruction pointer contains the linear address where execution is instructed to halt (see third argument of `init-x86-state` above).

   ```
   (x86-run <n> x86)
   ```

   The contents of the `ms` field after the termination of this run will indicate whether the program ran successfully or not. One can also dynamically debug the program, as described in Section 3.2 below. Upon the successful completion of the program, its output can be inspected by reading the relevant components in the final x86 state.

Note that if one needs to perform another concrete run of the program in the same ACL2 session, the x86 state must be initialized again — in particular, the instruction pointer must point to the first instruction to be executed and the `ms` field must be `nil`.

All these utilities aside, how does one determine the initial values of the components of the x86 state? A user familiar with x86 assembly and/or machine code may be able to figure this out simply by "reading" the program — this task may be easier if the high-level source program is available too. A possibly less time-consuming alternative is to run the program on the real machine (or an instruction-set simulator like QEMU [3]) and take a snapshot of the contents of registers, flags, memory locations, etc. immediately before the first instruction to be executed. The x86 model's state can be initialized with all of these values. This second approach has the benefit that the model's initial state matches that of the real machine, which helps in model validation via co-simulations.

The topic `:doc x86isa::program-execution` contains more information on initializing the x86 state. Examples of such `foo-run.lsp` files for various programs are also available online.

3.2 Monitoring a Concrete Run on the x86 ISA Model

Tools like the GDB (GNU Debugger) and Pintool are used to monitor x86 machine-code programs at runtime. GDB allows profiling by executing a program one instruction at a time, inserting breakpoints, etc., whereas Pintool injects instrumentation code into the program itself[7]. We provide some utilities in the `x86isa` library that mimic these tools; the capabilities currently provided are as follows:

- Stepping the interpreter once, à la `stepi` command of GDB;
- Stepping the interpreter `n` instructions at a time;
- Inserting breakpoints where the execution of the program will stop: Arbitrary ACL2 functions can be used to define these stopping points. An illustrative example is as follows: one can write an ACL2 function that computes the sum of values in a range of memory addresses and insert a breakpoint that instructs the x86 interpreter to halt as soon as the value returned by this function becomes greater than the current value of the `rax` register.
- Logging all memory read and write operations;

[7] It should be noted that instrumentation code, such as that included by Pintool, is supposed to be transparent to the program; however, it is not unexpected for such code to inadvertently alter the behavior of the program. Instead of injecting x86 machine code, our utilities monitor the ACL2 specification functions of our x86 model.

- Logging the x86 state (sans the memory) — either the current x86 state or the x86 state obtained after every instruction or after every breakpoint can be logged to a file.

The ability to log the x86 state is useful in co-simulations — one can simply compare the logs generated by the program to those by the model in order to perform model validation. Syntax and usage of our monitoring utilities are described at `:doc x86isa::dynamic-instrumentation`.

4 Formal Analysis of x86 Programs

Given an ACL2 machine model defined using operational semantics, various code proof styles can be used for program verification. We do not discuss them in this paper, and refer the reader elsewhere [8] for details. However, central to almost all these proof styles is the ability to symbolically simulate a program. The `x86isa` books provide the usual ACL2 rules that enable symbolic simulation of x86 machine-code programs by controlling the *unwinding* of the interpreter.

1. **Step Function Opener Rule:** This rule dictates the conditions under which a call of the step function, `x86-fetch-decode-execute`, should be expanded by ACL2. For instance, one of these conditions is that the `ms` field in the initial x86 state should be `nil`. Because of this rule, ACL2 first expands that call of the step function about which enough information to resolve the hypotheses of this rule is known. Typically, this means expanding the call corresponding to the current instruction (i.e., the instruction located at the address contained in the instruction pointer).
2. **Run Function Sequential Composition Rule:** This rule facilitates compositional reasoning by reducing the problem of reasoning about `(n1 + n2)` number of instructions to two smaller problems — first reasoning about `n1` instructions, and then about `n2` instructions. That is, it rewrites expressions of the form `(x86-run (clk+ n1 n2) x86)` to `(x86-run n2 (x86-run n1 x86))` when applicable.
3. **Run Function Opener Rule:** This rule controls the expansion of the run function by rewriting `(x86-run n x86)` to `(x86-run (1- n) (x86-fetch-decode-execute x86))`, provided that the `ms` field is `nil` and `n` is not equal to zero.

Additionally, the `x86isa` books also contain read-over-write and write-over-write rules that describe the interaction between the x86 state accessor and updater functions using the notions of non-interference and overlap. An example of a simple non-interference property is that a write to a register `i` does not interfere with a subsequent read from a register `j`, provided that `i` and `j` are distinct. Analogously, an example of a simple overlap property is that if consecutive writes are made to register `i`, then the most recent write will be the only visible one.

Thus, the `x86isa` books include the typical ACL2 rules that will be immediately familiar to a user with some experience in code proofs. These books provide lemma libraries to support almost completely automated symbolic simulation of many x86 programs — we have also documented how a user can debug a failed attempt at unwinding the x86 interpreter (`:doc x86isa::debugging-code-proofs`). We now give some examples of reasoning about x86 machine-code programs as a way to illustrate the different methodologies a user can adopt when working with the `x86isa` books.

4.1 Verifying Application Programs

The user-level mode of operation of the x86 ISA model is well-suited to application program verification, due to reasons discussed previously in Section 2.2. We consider the following two kinds of

application programs here: the first is structurally simple and contains straight-line code that performs dense arithmetic and logical operations on fixed-width inputs (e.g., sub-routines that do bit twiddling), and the second contains loops, branches, and maybe even makes some system calls (e.g., a sub-routine that computes the word-count of a file).

Using the x86isa books, one can choose to verify the first program completely automatically — without using any rules provided by the x86isa books — by using the bit-blasting capabilities of GL [15, 16] but at the expense of a general theorem of correctness. That is, one may need to constrain the initial x86 state by assigning concrete values instead of symbolic ones to certain components in order to reduce the complexity of the AIGs generated by GL, thereby making the problem tractable for bit-blasting. An example of such a theorem is x86-popcount-64-correct below, which states that a given program that is intended to compute the population count of its 64-bit input n meets its specification (proof script and a detailed description [11] are available). This theorem is in terms of the program being located at fixed addresses instead of being (mostly) position-independent[8] — note that *popcount-64* is a list of pairs of fixed linear addresses and the program's bytes.

```
(defconst *popcount-64*
  (list
   (cons #x400650 #x89) ;; mov %edi,%edx
   (cons #x400651 #xfa)
   ;; ...                ... many instructions elided ...
   (cons #x4006c2 #xc3) ;; retq
   ))

(def-gl-thm x86-popcount-64-correct
  :hyp (and (natp n)
            (< n (expt 2 64)))
  :concl
  (b* ((start-address #x400650)
       (halt-address #x4006c2)
       ;; Assigning default values to state components:
       (x86 (create-x86))
       (x86 (!user-level-mode t x86))
       ((mv flg x86)
        (init-x86-state
         nil start-address halt-address
         nil nil nil 0 *popcount-64* x86))
       ;; Input n is symbolic and located in rdi.
       (x86 (wr64 *rdi* n x86))
       ;; 300 is the upper bound on the number of steps to take.
       ;; Execution halts if the halt-address is reached earlier.
       (x86 (x86-run 300 x86)))
    (and
     ;; No error was encountered during state initialization.
     (equal flg nil)
```

[8] We say "mostly" because a program's location, even a parameterized one, needs to be constrained in some important ways so that it does not overlap with the stack, data, etc.

```
    ;; rax contains the popcount of n.
    (equal (rgfi *rax* x86) (logcount n))
    ;; rip contains the address of the instruction following the
    ;; halt address.
    (equal (rip x86) (+ 1 halt-address))))
  :g-bindings
  `((n   (:g-number ,(gl-int 0 1 65)))))
```

Of course, such a theorem is not useful if re-compiling the same high-level program results in a different machine-code program, which may be located at a different linear memory location or may use different x86 instructions altogether. The former case would not have been a problem if `x86-popcount-64-correct` was a statement about the position-independent version of the program, whereas the latter case is a downside of machine-code verification in general. The benefit of this approach is that it provides a simple solution in both these situations. A user can simply re-submit the new conjecture to ACL2 so that the proof proceeds completely automatically as before.

GL may reach its limits if used to verify the second kind of program, which is structurally complex and whose proof involves determining inductive invariants. One can reason about this program on the x86 ISA model using the classical *clock functions* approach [9]. A clock function computes the number of steps needed for the program to reach a desired state. The program's correctness can be stated as follows: given a state $\texttt{x86}_\text{i}$ that satisfies some preconditions, the final state $\texttt{x86}_\text{f}$ `= x86-run(n,` $\texttt{x86}_\text{i}$`)`, where `n` denotes the (possibly symbolic) value computed by the clock function, satisfies some postconditions. For instance, we verified a word-count program that makes system calls to read input from a file — the proof script and a detailed description [14] are available. One of the final theorems of correctness for this program is as follows. The function `preconditions` specifies general conditions for the correctness of this program — e.g., the program is located at a suitable symbolic address `addr` in the initial x86 state, whose various components contain symbolic values.

```
(defthm program-behavior-nc
  (implies
   (and (preconditions addr x86)
        (equal offset (offset x86))
        (equal str-bytes (input x86)))
   (equal
    ;; nc: gets the number of characters computed by the program
    ;; from the x86 state
    (nc (x86-run (clock str-bytes x86) x86))
    ;; nc-spec: specification function that computes the number
    ;; of characters
    (nc-spec offset str-bytes))))
```

Though the amount of user interaction required is higher in this case, one obtains a more general correctness theorem than that for the first program. The applicability of this theorem is still adversely affected in case the machine-code program generated by re-compiling the source program contains different x86 instructions. However, it is possible that one may be able to re-use some lemmas from the proof script.

In addition to functional correctness, the `x86isa` books provide lemmas that help in the proof of other kinds of properties. For instance, for the word-count program, we proved that the values in all memory locations, except the program's stack, in the final x86 state are exactly the same as that in the initial state. In other words, this theorem states that at the end of its execution, the word-count program did not change any values in unintended regions of memory.

We note that the position-independent version of the popcount program can also be verified using the clock functions approach without incurring too much overhead. The lemmas supporting symbolic simulation in the x86isa books can help in automatically obtaining the ACL2 expression corresponding to the value in the rax register in the final x86 state, and GL can be used to prove that this expression computes the same value as that computed by logcount. This ACL2 expression obtained after symbolic simulation will change if the instructions in the machine program change due to re-compilation of the source code. In this case too, one can use GL to automatically prove the equality of this expression with logcount. Thus, not only does this approach win us a general theorem of correctness, but it also provides a relatively cheap way of re-proving a general correctness theorem in case the program changes. In this manner, GL can be used to reason about computationally intensive pieces of code in a structurally complex program, and lemmas provided by the x86isa books can be used to perform compositional reasoning to obtain the proof of correctness of the entire program. More details can be found along with the popcount proof script.

4.2 Verifying System Programs

System programs can also be verified using the same strategies as application programs. However, generally speaking, the reasoning overhead of system programs is higher because they access and modify a larger x86 state than application programs. In this paper, we focus only on the upshot of the processor's side-effect updates to *accessed* and *dirty flags* during address translation via paging.

The accessed and dirty flags are two fields present in the entries of the paging data structures. Whenever an entry is referenced during an address translation, the processor sets its accessed flag. When the translation is done on behalf of a memory write operation, then the processor sets the dirty flag in the final entry that points to the physical address. Effectively, these updates are side-effects of the processor as it works to translate a linear address. Thus, all linear memory operations — including memory reads — modify the memory, as a result of which one needs to establish non-interference (or overlap) properties about every linear memory operation. These side-effect updates are numerous — merely fetching one byte of an x86 instruction from the memory can cause many side-effect updates. The sheer number of these side-effect updates increases reasoning overhead significantly.

The system-level mode of operation offers two sub-modes — marking and non-marking mode — that are exactly the same except in their treatment of these side-effect updates. The marking mode of operation specifies these side-effect updates to the accessed and dirty flags, whereas the non-marking mode suppresses them. For supervisor-mode programs that do not depend on these side-effect updates, we recommend verifying the program in the non-marking mode and then porting the proof over to the marking mode. This is because of the simpler linear memory non-interference theorems in the non-marking mode — these theorems have fewer hypotheses in the non-marking mode because one does not have to preclude reads from those regions of the memory that are changed by the side-effect updates. Porting the proof over to the marking mode is simply a matter of adding relevant (and mostly obvious) disjointness preconditions to the theorems — for example, the paging entries that govern the translation of the program and the program itself must be disjoint. Note that this condition was unnecessary in the non-marking mode because the paging entries of the program are not modified over the course of its execution. The x86isa library contains books that includes a proof of correctness of a supervisor-mode zero-copy program that manipulates the paging data structures; this proof illustrates this methodology of first using the non-marking mode and then the marking mode to verify system programs.

5 How Do I...?

We anticipate and answer some specific how-to questions that may be asked by a user and potential future contributor to the x86isa books.

How do I add a new component to the x86 state?
The x86 state is defined using an abstract stobj [12] layered on top of a concrete stobj [7]. The "default" abstract representation in our model for simple fields is the same as the logical representation of the concrete field; for array fields, it is a record [5] with a default value of 0.

Suppose one wanted to add the (currently missing) 64-bit Extended Control Register xcr0 to the x86 state. One must first add a suitable field to the concrete stobj x86$c. Note that xcr0$c is a simple (non-array) field.

```
(xcr0$c :type (unsigned-byte 64) :initially 0)
```

The x86isa books contain macros that use the above expression to generate suitable events that will help in defining and admitting the corresponding abstract stobj. For instance, the abstract and top-level recognizer, accessor, and updater functions, along with the correspondence function pertaining to this field will be automatically generated. One would have to resolve the proof obligations that establish the correspondence between the concrete and abstract stobjs (these obligations are generated automatically by the defabsstobj event), but these will be straightforward for "default" abstract representations.

If a different abstract representation for the new component is required, one would have to disable the automated generation of events for this component and manually define the appropriate events. The memory model in our x86 state is an example of where we used this manual approach — the correspondence relation between the concrete and abstract representation of the x86 memory is complicated. Memory is implemented using accessor, updater, and recognizer functions operating on three distinct fields in the concrete stobj, and these functions have been proved to correspond to those operating on a single record field in the abstract stobj. See [12, 17] for details.

We now discuss a modeling quirk of MSRs (machine-specific registers), since it is likely that a user might want to add more MSRs than are currently supported by our x86 model. The x86 ISA defines several architecture-specific MSRs (possibly, hundreds), but we model only six of them using an array field in the concrete stobj and a corresponding record in the abstract stobj. In order to add a new MSR, simply increase the number of elements of the msr$c field. The caveat here is that unlike other registers, the index of an MSR in our x86 state does not match its identifier on the real machine. For example, the 0^{th} general-purpose register in the x86 state of our model is the rax, and 0 is also its identifier on the real machine. It suffices to define one constant — *rax* = 0 — pertaining to this register. However, the 0^{th} MSR in our x86 state corresponds to the ia32_efer register, whose identifier on the real machine is 0xC000_0080. Thus, we define two constants pertaining to MSRs: one for the real identifier and one for indexing into the msr field of our model. For instance, *ia32_efer* is equal to 0xC000_0080 whereas *ia32_efer-idx* is equal to 0. If an x86 instruction uses an identifier equal to *ia32_efer*, our specifications use *ia32_efer-idx* to access this register.

How do I add a new x86 instruction?
There are four main tasks here:

1. *Add the capability to decode the instruction:* The Intel manuals [1] have various tables that contain the decoding information of x86 instructions, such as their addressing information (e.g., whether the instruction uses a ModR/M byte to determine the location of its operands), default sizes, etc.

The x86isa books have the Lisp/ACL2 representation of these tables (see this book) — any instruction with a one- or two-byte x86 opcode can be decoded using our ACL2 functions that read information off these tables. For all other instructions (e.g., those that have three-byte opcodes), one would have to manually port some more relevant tables from the Intel manuals into the x86isa books.

2. *Write the instruction semantic function:* The x86isa books supply a def-inst macro to specify instruction semantic functions. This macro is simply a wrapper around define — it also adds the instruction's details to a table automatically so that one can keep track of the opcodes supported by the x86isa books.
3. *Extend the step function:* Depending on the opcode of an instruction, the step function x86-fetch-decode-execute dispatches control to an appropriate instruction semantic function. Simply invoke the new instruction semantic function from the step function.
4. *Validate the instruction's specification:* Using utilities described in Section 3, run co-simulations of the model against the real machine to validate the new instruction semantic function.

How do I abstract away the behavior of a standard C library function, say printf **or** scanf**?**
As described earlier, the user-level mode of operation extends the semantics of the syscall instruction to support both reasoning and execution of some OS-provided system calls. Standard C library functions like printf and scanf are built on top of these system calls. One may want to assume the correctness of these library functions instead of the lower-level system calls when reasoning about an application program.

One solution is to create yet another mode of operation of the x86 ISA model — say, a *strong* user-level mode — where the semantics of call, jmp, and any other branch/control-flow instruction have been extended to support these standard library functions. Note that similar to system calls, these functions will be non-deterministic from the point of view of the application. Thus, one can use the env field to model the external environment that these library functions depend on.

6 Potential Future Projects

Though the x86isa books model a significant portion of the x86-64 ISA, they are incomplete. Also, there are several ways in which its lemma libraries can be improved. We now discuss some short- and long-term projects that, once completed, can improve the quality and feature-set of x86isa. Of course, this list is non-exhaustive.

ISA Modeling Projects

- ***Exceptions and Interrupts:*** As of this writing, the x86isa books model system registers relevant to both exceptions and interrupts (idtr, gdtr, ldtr, and tr) and contain a specification of the Interrupt, Global, and Local Descriptor Tables (IDT/GDT/LDT), including recognizers for well-formed table entries or *gates*. These gates contain information about the location of the interrupt- or exception-handling procedures. The x86isa books also support system instructions like lgdt, lldt, and lidt used to initialize the system registers.

  We already support exceptions in a limited sense — whenever we detect that an error condition is reached (for example, if a div instruction's divisor operand is 0, which corresponds to a #DE

exception; or if a paging entry encountered during a page walk is ill-formed, which corresponds to a `#PF` exception), we populate the `ms` field with some information relevant to this exception and halt the interpreter. Currently, the preconditions for the correctness of programs analyzed using `x86isa` weed out all such erroneous conditions. In order to support exceptions in their entirety, the appropriate exception-handling procedures must be called by looking up relevant information in the descriptor tables — the current solution of populating the `ms` is only a stopgap. On the other hand, interrupts are asynchronous events (unlike exceptions) and their implementation is likely to require some significant changes/additions to the x86 ISA model. For example, since all interrupts are guaranteed to be taken on an instruction boundary, we could consult an oracle to check for the occurrence of some interrupt at every such boundary. Upon encountering one, we can transfer control to the appropriate interrupt-handling procedure. Note that one would need to modify the step and/or the run functions if we adopt this approach.

Given that much of the support required for implementing both exceptions and interrupts already exists in the x86 model, we estimate that this will be a relatively short-term project.

- ***I/O Capabilities:*** I/O instructions like `in` and `out` are not supported by `x86isa` yet. Implementing them will also be a short-term project because the existing infrastructure around `env` can be re-used to characterize interaction of the processor with an external environment.
- ***Caches and Multiprocessors:*** Our x86 model can be extended to include the entire memory hierarchy — including caches, translation-look aside buffers, store buffers, etc. — in order to obtain a complete specification of how memory reads and writes are resolved by multiple processors. This promises to be quite a long-term project because it would involve dealing with complicated properties like cache coherence.

We briefly comment on a challenge that a contributor to the `x86isa` books is likely to face when modeling some advanced features of the x86 ISA, such as interrupts or protection management. Ideally, to validate the x86 ISA model, one must co-simulate it against a "bare" x86 processor, i.e., one not running any OS. However, a bare x86 machine is difficult to work with, and so far we have validated our model against an x86 ISA processor running a mainstream OS like Linux[9]. Unfortunately, an OS is tightly inter-twined with the workings of a processor's system features, thereby making it difficult to separate OS-specific behavior from the machine's behavior. We posit that a satisfactory way to validate the specification of system features is by running a mainstream (if stripped down) OS on the x86 ISA model. This task will require adding several features and instructions currently missing from `x86isa`. Not only will this undertaking increase confidence in the accuracy of the x86 model, but it will also enable us to reason about *real* system code that is deployed on modern machines. Needless to say, this is a formidable long-term project, but one with high returns.

Proof-related Projects

- ***Using Codewalker:*** One can imagine using Codewalker to lift reasoning about x86 machine code to high-level specification functions. This project may involve adding capabilities to the Codewalker and/or `x86isa` books.
- ***Automated Precondition Discovery:*** A challenging aspect of program verification is discovering the preconditions under which a program behaves as expected. One way in which ACL2 users figure out some of the preconditions is by observing the reason why some rules fail to fire when

[9] Of course, one could also choose to validate the x86 ISA model against a hardened instruction-set simulator like QEMU.

expected, and then adding any missing hypotheses to the conjecture that make those rules applicable to the goal at hand. A useful project could be to automate this process so that after a failed proof attempt, a user is presented with hypotheses that are good candidates to be top-level preconditions. One can imagine such a capability to be useful in other ACL2 projects too. As such, this project can easily be a long-term one.

7 Conclusion

This paper serves as a good starting point for a user or a potential future developer of the x86isa books. We recommend that a new user and/or someone with little experience in low-level code verification begin by executing some concrete runs of a program on the x86 model before moving on to program verification. Also, it is advisable — for both reasoning and execution — to first use the x86 model in its user-level mode instead of the more complicated system-level mode of operation.

We give an overview of the development style and capabilities of the x86isa books in this paper. However, a more complete description is available [10] — this work's accompanying Ph.D. dissertation describes how the x86 model is optimized for both reasoning and execution efficiency, how complicated x86 ISA mechanisms such as IA-32e paging and segmentation are specified, how congruence-based rewriting is used to reduce reasoning overhead in the system-level mode of operation, and other pertinent details. Also, the most up-to-date information about these books is available at `:doc x86isa`.

There are many ways in which these books can be used and/or extended, beyond what we discussed in the previous section. An example of one such application of this work, not related to program verification, is micro-architecture verification — e.g., one can use the x86 ISA model to prove that one or more x86 micro-operations implement an ISA-level instruction. This work paves the way for research and engineering opportunities that would otherwise have been difficult to pursue — we hope that the community gets involved in this project.

Acknowledgements

This work was supported by DARPA under contract number N66001-10-2-4087, and by NSF under contract number CNS-1525472. I am grateful to Warren Hunt, Jr., Matt Kaufmann, J Moore, and Robert Watson for their guidance. I thank Eric Smith for helpful discussions about his use of and expectations from the x86isa books, and Anna Slobodova for feedback on an early version of this paper.

References

[1] *Intel 64 and IA-32 Architectures Software Developer's Manuals.* Online. Order Number: 325462-059US. (June 2016).
http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[2] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood (2005): *Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation*. *SIGPLAN Not.* 40(6), pp. 190–200, doi:10.1145/1064978.1065034.

[3] Fabrice Bellard (2005): *QEMU, a Fast and Portable Dynamic Translator*. In: *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*, pp. 41–46. Available at http://www.usenix.org/events/usenix05/tech/freenix/bellard.html.

[4] *OS X ABI Mach-O File Format Reference*. Online; accessed: January 2017. https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/MachORuntime/index.html.

[5] Matt Kaufmann and Rob Sumners: *Efficient Rewriting of Operations on Finite Structures in ACL2*. In: *Proceedings of 3rd International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2002)*.

[6] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell (2005): *Chapter 4: Object Files in System V Application Binary Interface*. *AMD64 Architecture Processor Supplement, Draft v0* 99.

[7] Robert S. Boyer and J S. Moore (2002): *Single-Threaded Objects in ACL2*. In: *Practical Aspects of Declarative Languages, 4th International Symposium, PADL 2002, Portland, OR, USA, January 19-20, 2002, Proceedings*, pp. 9–27, doi:10.1007/3-540-45587-6_3.

[8] Sandip Ray and J S. Moore (2004): *Proof Styles in Operational Semantics*. In: *Formal Methods in Computer-Aided Design, 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004, Proceedings*, pp. 67–81, doi:10.1007/978-3-540-30494-4_6.

[9] Sandip Ray, Warren A. Hunt Jr., John Matthews, and J S. Moore (2008): *A Mechanical Analysis of Program Verification Strategies*. *J. Autom. Reasoning* 40(4), pp. 245–269, doi:10.1007/s10817-008-9098-1.

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

[11] Shilpi Goel and Warren A. Hunt, Jr. (2014): *Automated Code Proofs on a Formal Model of the x86*. In: *Verified Software: Theories, Tools, Experiments (VSTTE'13)*, *Lecture Notes in Computer Science* 8164, Springer Berlin Heidelberg, pp. 222–241., doi:10.1007/978-3-642-54108-7_12.

[12] Shilpi Goel, Warren A. Hunt Jr. and Matt Kaufmann (2013): *Abstract Stobjs and Their Application to ISA Modeling*. In: *Proceedings International Workshop on the ACL2 Theorem Prover and its Applications, ACL2 2013, Laramie, Wyoming, USA, May 30-31, 2013.*, pp. 54–69, doi:10.4204/EPTCS.114.5.

[13] Shilpi Goel, Warren A. Hunt, Jr., and Matt Kaufmann (2017): *Engineering a Formal, Executable x86 ISA Simulator for Software Verification*, pp. 173–209. Springer International Publishing, Cham, doi:10.1007/978-3-319-48628-4_8.

[14] Shilpi Goel, Warren A. Hunt Jr., Matt Kaufmann, Soumava Ghosh (2014): *Simulation and Formal Verification of x86 Machine-code Programs that Make System Calls*. In: *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*, pp. 91–98, doi:10.1109/FMCAD.2014.6987600.

[15] Sol Swords (2010): *A Verified Framework for Symbolic Execution in the ACL2 Theorem Prover*. Ph.D. thesis, Department of Computer Science, The University of Texas at Austin. Available at http://hdl.handle.net/2152/ETD-UT-2010-12-2210.

[16] Sol Swords and Jared Davis (2011): *Bit-Blasting ACL2 Theorems*. In: *Proceedings of the 10th International Workshop on the ACL2 Theorem Prover and its Applications, ACL2 2011, Austin, Texas, USA, November 3-4, 2011.*, pp. 84–102, doi:10.4204/EPTCS.70.7.

[17] Warren A. Hunt Jr. and Matt Kaufmann (2012): *A Formal Model of a Large Memory that Supports Efficient Execution*. In: *Formal Methods in Computer-Aided Design, FMCAD 2012, Cambridge, UK, October 22-25, 2012*, pp. 60–67. Available at http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6462556.

The Cayley-Dickson Construction in ACL2

John Cowles Ruben Gamboa
University of Wyoming
Laramie, WY
{cowles,ruben}@uwyo.edu

The **Cayley-Dickson Construction** is a generalization of the familiar construction of the complex numbers from pairs of real numbers. The complex numbers can be viewed as two-dimensional vectors equipped with a multiplication.

The construction can be used to construct, not only the two-dimensional Complex Numbers, but also the four-dimensional Quaternions and the eight-dimensional Octonions. Each of these vector spaces has a vector multiplication, $\mathbf{v}_1 \bullet \mathbf{v}_2$, that satisfies:

1. Each nonzero vector, $\mathbf{v}$, has a multiplicative inverse $\mathbf{v}^{-1}$.
2. For the Euclidean length of a vector $|\mathbf{v}|$, $|\mathbf{v}_1 \bullet \mathbf{v}_2| = |\mathbf{v}_1| \cdot |\mathbf{v}_2|$

Real numbers can also be viewed as (one-dimensional) vectors with the above two properties.

ACL2(r) is used to explore this question: Given a vector space, equipped with a multiplication, satisfying the Euclidean length condition 2, given above. Make pairs of vectors into "new" vectors with a multiplication. When do the newly constructed vectors also satisfy condition 2?

1 Cayley-Dickson Construction

Given a vector space, with vector addition, $\mathbf{v}_1 + \mathbf{v}_2$; vector minus $-\mathbf{v}$; a zero vector $\vec{0}$; scalar multiplication by real number a, $a \circ \mathbf{v}$; a unit vector $\vec{1}$; and vector multiplication $\mathbf{v}_1 \bullet \mathbf{v}_2$; satisfying the Euclidean length condition $|\mathbf{v}_1 \bullet \mathbf{v}_2| = |\mathbf{v}_1| \cdot |\mathbf{v}_2|$ (2).

Define the **norm** of vector $\mathbf{v}$ by $\|\mathbf{v}\| = |\mathbf{v}|^2$. Since $|\mathbf{v}_1 \bullet \mathbf{v}_2| = |\mathbf{v}_1| \cdot |\mathbf{v}_2|$ is equivalent to $|\mathbf{v}_1 \bullet \mathbf{v}_2|^2 = |\mathbf{v}_1|^2 \cdot |\mathbf{v}_2|^2$, the Euclidean length condition is equivalent to

$$\|\mathbf{v}_1 \bullet \mathbf{v}_2\| = \|\mathbf{v}_1\| \cdot \|\mathbf{v}_2\|.$$

Recall the **dot (or inner) product**, of n-dimensional vectors, is defined by

$$\begin{aligned}(x_1,\ldots,x_n) \odot (y_1,\ldots,y_n) &= x_1 \cdot y_1 + \cdots + x_n \cdot y_n \\ &= \sum_{i=1}^{n} x_i \cdot y_i\end{aligned}$$

Then Euclidean length and norm of vector $\mathbf{v}$ are given by

$$\begin{aligned}|\mathbf{v}| &= \sqrt{\mathbf{v} \odot \mathbf{v}} \\ \|\mathbf{v}\| &= \mathbf{v} \odot \mathbf{v}.\end{aligned}$$

Except for vector multiplication, it is easy to treat ordered pairs of vectors, $(\mathbf{v}_1;\mathbf{v}_2)$, as vectors:

1. $(\mathbf{v}_1;\mathbf{v}_2) + (\mathbf{w}_1;\mathbf{w}_2) = (\mathbf{v}_1 + \mathbf{w}_1;\mathbf{v}_2 + \mathbf{w}_2)$
2. $-(\mathbf{v}_1;\mathbf{v}_2) = (-\mathbf{v}_1;-\mathbf{v}_2)$

Warren Hunt Jr. and Anna Slobodova (Eds.):
ACL2 Theorem Prover and its Applications
EPTCS 249, 2017, pp. 18–29, doi:10.4204/EPTCS.249.2

3. zero vector: $(\vec{0};\vec{0})$
4. $a \circ (\mathbf{v}_1;\mathbf{v}_2) = (a \circ \mathbf{v}_1; a \circ \mathbf{v}_2)$
5. unit vector: $(\vec{1};\vec{0})$
6. $(\mathbf{v}_1;\mathbf{v}_2) \odot (\mathbf{w}_1;\mathbf{w}_2) = \{\mathbf{v}_1 \odot \mathbf{w}_1\} + \{\mathbf{v}_2 \odot \mathbf{w}_2\}$
7. $\|(\mathbf{v}_1;\mathbf{v}_2)\| = \|\mathbf{v}_1\| + \|\mathbf{v}_2\|$

Given that

$$\|\mathbf{v} \bullet \mathbf{w}\| = \|\mathbf{v}\| \cdot \|\mathbf{w}\|,$$

the problem is to define **multiplication of vector pairs** so that

$$\|(\mathbf{v}_1;\mathbf{v}_2) \bullet (\mathbf{w}_1;\mathbf{w}_2)\| = \|(\mathbf{v}_1;\mathbf{v}_2)\| \cdot \|(\mathbf{w}_1;\mathbf{w}_2)\|.$$

1.1 Examples

Complex multiplication. Think of the real numbers as one-dimensional vectors. Interpret ordered pairs of real numbers as complex numbers: For real a and b, $(a;b) = (\mathsf{complex}\ a\ \ b) = a + b \cdot i$. For reals a_1, a_2 and b_1, b_2, complex multiplication is defined by

$$(a_1;a_2) \cdot (b_1;b_2) = ([a_1b_1 - a_2b_2]; [a_1b_2 + a_2b_1]) \tag{1}$$

and satisfies

$$\|(a_1;a_2) \cdot (b_1;b_2)\| = \|(a_1;a_2)\| \cdot \|(b_1;b_2)\|.$$

Quaternion multiplication. Think of the complex numbers as two-dimensional vectors. Interpret ordered pairs of complex numbers as William Hamilton's quaternions [1, 3, 2]: For complex $c = a_1 + a_2 \cdot i$ and $d = b_1 + b_2 \cdot i$,

$$\begin{aligned} (c;d) &= c + d \cdot j \\ &= a_1 + a_2 \cdot i + b_1 \cdot j + b_2 \cdot ij \\ &= a_1 + a_2 \cdot i + b_1 \cdot j + b_2 \cdot k. \end{aligned}$$

where $i \cdot j = k$.

For complex $c = a_1 + a_2 \cdot i$, $\bar{c} = a_1 - a_2 \cdot i$ is the **conjugate** of c.

For complex numbers $c_1 = a_1 + a_2 \cdot i$, $c_2 = a_3 + a_4 \cdot i$, $d_1 = b_1 + b_2 \cdot i$, and $d_2 = b_3 + b_4 \cdot i$, quaternion multiplication is defined by

$$\begin{aligned} (c_1;c_2) \cdot (d_1;d_2) &= (a_1b_1 - a_2b_2 - a_3b_3 - a_4b_4) \\ &\quad + (a_1b_2 + a_2b_1 + a_3b_4 - a_4b_3)i \\ &\quad + (a_1b_3 - a_2b_4 + a_3b_1 + a_4b_2)j \\ &\quad + (a_1b_4 + a_2b_3 - a_3b_2 + a_4b_1)k \\ &= (a_1b_1 - a_2b_2) + (a_1b_2 + a_2b_1)i \\ &\quad - [(a_3b_3 + a_4b_4) + (-a_3b_4 + a_4b_3)i] \\ &\quad + [(a_1b_3 - a_2b_4) + (a_1b_4 + a_2b_3)i]j \\ &\quad + [(a_3b_1 + a_4b_2) + (-a_3b_2 + a_4b_1)i]j \\ &= (c_1d_1 - c_2\bar{d_2}) + (c_1d_2 + c_2\bar{d_1})j \\ &= ([c_1d_1 - c_2\bar{d_2}]; [c_1d_2 + c_2\bar{d_1}]) \end{aligned} \tag{2}$$

and satisfies

$$\|(c_1;c_2)\cdot(d_1;d_2)\| = \|(c_1;c_2)\|\cdot\|(d_1;d_2)\|.$$

Quaternion multiplication is completely determined by this table for the multiplication of i, j, and k:

	i	j	k
i	-1	k	$-j$
j	$-k$	-1	i
k	j	$-i$	-1

Quaternion multiplication is not commutative, since $i\cdot j = k$ and $j\cdot i = -k$. Quaternion multiplication is associative: $(q_1\cdot q_2)\cdot q_3 = q_1\cdot(q_2\cdot q_3)$.

Octonion multiplication. Think of the quaternions as four-dimensional vectors. Interpret ordered pairs of quaternions as John Graves's and Arthur Cayley's octonions [1, 3, 2]: For quaternions $q = a_1 + a_2\cdot i + a_3\cdot j + a_4\cdot k$ and $r = b_1 + b_2\cdot i + b_3\cdot j + b_4\cdot k$,

$$\begin{aligned}(q;r) &= q + r\cdot\ell \\ &= a_1 + a_2\cdot i + a_3\cdot j + a_4\cdot k \\ &\quad + b_1\cdot\ell + b_2\cdot i\ell + b_3\cdot j\ell + b_4\cdot k\ell \\ &= a_1 + a_2\cdot i + a_3\cdot j + a_4\cdot k \\ &\quad + b_1\cdot\ell + b_2\cdot I + b_3\cdot J + b_4\cdot K.\end{aligned}$$

where $i\cdot\ell = I$, $j\cdot\ell = J$, and $k\cdot\ell = K$

For quaternion $q = a_1 + a_2\cdot i + a_3\cdot j + a_4\cdot k$, $\bar{q} = a_1 - a_2\cdot i - a_3\cdot j - a_4\cdot k$ is the **conjugate** of q.

For quaternions q_1, q_2, r_1, and r_2, octonion multiplication is defined by

$$\begin{aligned}(q_1;q_2)\cdot(r_1;r_2) &= (q_1 + q_2\cdot\ell)\cdot(r_1 + r_2\cdot\ell) \\ &= (q_1r_1 - \bar{r}_2q_2) + (r_2q_1 + q_2\bar{r}_1)\cdot\ell \\ &= ([q_1r_1 - \bar{r}_2q_2];[r_2q_1 + q_2\bar{r}_1]) \qquad (3)\end{aligned}$$

and satisfies

$$\|(q_1;q_2)\cdot(r_1;r_2)\| = \|(q_1;q_2)\|\cdot\|(r_1;r_2)\|.$$

Octonion multiplication is completely determined by this table for the multiplication of i, j, k, ℓ, I, J, and K:

	i	j	k	ℓ	I	J	K
i	-1	k	$-j$	I	$-\ell$	$-K$	J
j	$-k$	-1	i	J	K	$-\ell$	$-I$
k	j	$-i$	-1	K	$-J$	I	$-\ell$
ℓ	$-I$	$-J$	$-K$	-1	i	j	k
I	ℓ	$-K$	J	$-i$	-1	$-k$	j
J	K	ℓ	$-I$	$-j$	k	-1	$-i$
K	$-J$	I	ℓ	$-k$	$-j$	i	-1

Since the octonions contain the quaternions, octonion multiplication is also not commutative. Octonian multiplication is not associative, since $\ell\cdot(I\cdot J) = K$ and $(\ell\cdot I)\cdot J = -K$.

Complex (1), quaternion (2), and octonion (3) multiplication suggest these possible definitions for vector multiplication:

$$(\mathbf{v}_1;\mathbf{v}_2)\bullet(\mathbf{w}_1;\mathbf{w}_2) = ([\mathbf{v}_1\mathbf{w}_1-\mathbf{v}_2\mathbf{w}_2];[\mathbf{v}_1\mathbf{w}_2+\mathbf{v}_2\mathbf{w}_1]) \quad (4)$$

$$(\mathbf{v}_1;\mathbf{v}_2)\bullet(\mathbf{w}_1;\mathbf{w}_2) = ([\mathbf{v}_1\mathbf{w}_1-\mathbf{v}_2\bar{\mathbf{w}}_2];[\mathbf{v}_1\mathbf{w}_2+\mathbf{v}_2\bar{\mathbf{w}}_1]) \quad (5)$$

$$(\mathbf{v}_1;\mathbf{v}_2)\bullet(\mathbf{w}_1;\mathbf{w}_2) = ([\mathbf{v}_1\mathbf{w}_1-\bar{\mathbf{w}}_2\mathbf{v}_2];[\mathbf{w}_2\mathbf{v}_1+\mathbf{v}_2\bar{\mathbf{w}}_1]) \quad (6)$$

When the $\mathbf{v}_i$ and $\mathbf{w}_i$ are real numbers, all three definitions are equivalent, since real multiplication is commutative and the conjugate of real a, $\bar{a}$, is just a.

When the $\mathbf{v}_i$ and $\mathbf{w}_i$ are complex numbers, the last two definitions are equivalent, since complex multiplication is commutative. However, when $\mathbf{v}_1 = 1+0\cdot i = \mathbf{w}_1$, $\mathbf{v}_2 = 0+1\cdot i$, and $\mathbf{w}_2 = 0+(-1\cdot i)$, the product given by equation (4) is the zero vector $([0+0\cdot i];[0+0\cdot i])$. So a vector product defined by (4) need not satisfy

$$\|(\mathbf{v}_1;\mathbf{v}_2)\bullet(\mathbf{w}_1;\mathbf{w}_2)\| = \|(\mathbf{v}_1;\mathbf{v}_2)\|\cdot\|(\mathbf{w}_1;\mathbf{w}_2)\|,$$

for complex inputs.

Since the quaternions contain the complex numbers, a vector product defined by (4) also will not be satisfactory for quaternion inputs. When $\mathbf{v}_1 = 0+(-1\cdot i)+0\cdot j+0\cdot k$, $\mathbf{v}_2 = 0+0\cdot i+1\cdot j+0\cdot k = \mathbf{w}_2$, and $\mathbf{w}_1 = 0+1\cdot i+0\cdot j+0\cdot k$, the product given by equation (5) is the zero vector $([0+0\cdot i+0\cdot j+0\cdot k];[0+0\cdot i+0\cdot j+0\cdot k])$. So a vector product defined by either (4) or (5) need not satisfy

$$\|(\mathbf{v}_1;\mathbf{v}_2)\bullet(\mathbf{w}_1;\mathbf{w}_2)\| = \|(\mathbf{v}_1;\mathbf{v}_2)\|\cdot\|(\mathbf{w}_1;\mathbf{w}_2)\|,$$

for quaternion inputs.

This leaves (6) as a possible way to define vector multiplication of ordered pairs of vectors. So a vector **conjugate**, $\bar{\mathbf{v}}$, is also required. Continuing the enumeration of vector pair operations given on page 2:

8. $\overline{(\mathbf{v}_1;\mathbf{v}_2)} = (\bar{\mathbf{v}}_1;-\mathbf{v}_2)$
9. $(\mathbf{v}_1;\mathbf{v}_2)\bullet(\mathbf{w}_1;\mathbf{w}_2) = ([\mathbf{v}_1\mathbf{w}_1-\bar{\mathbf{w}}_2\mathbf{v}_2];[\mathbf{w}_2\mathbf{v}_1+\mathbf{v}_2\bar{\mathbf{w}}_1])$

The enumerated list of items, $1,2,\cdots,9$, defining various operations on ordered pairs of vectors, is called the **Cayley-Dickson Construction** [4, 2].

1.2 Summary

- Start with the real numbers.
  Apply the Cayley-Dickson Construction to pairs of real numbers.
  Obtain a vector algebra isomorphic to the complex numbers.
- Apply the Cayley-Dickson Construction to pairs of complex numbers.
  Obtain a vector algebra isomorphic to the quaternions.
- Apply the Cayley-Dickson Construction to pairs of quaternions.
  Obtain a vector algebra isomorphic to the octonians.

Each of these vector spaces: real numbers, complex numbers, quaternions, and octonians, satisfy

$$\|\mathbf{v}_1\bullet\mathbf{v}_2\| = \|\mathbf{v}_1\|\cdot\|\mathbf{v}_2\|.$$

Futhermore, every vector $\mathbf{v}\neq\vec{0}$ has a multiplicative inverse:

$$\mathbf{v}^{-1} = \frac{1}{\|\mathbf{v}\|}\circ\bar{\mathbf{v}}$$

2 Composition Algebras

A **composition algebra** [1, 3] is a real vector space, with vector multiplication, a real-valued norm, and a real-valued dot product, satisfying this composition law

$$\|\mathbf{v}_1 \bullet \mathbf{v}_2\| = \|\mathbf{v}_1\| \cdot \|\mathbf{v}_2\|.$$

Use `encapsulate` to axiomatize, in ACL2(r), composition algebras.
ACL2(r) function symbols are needed for

- Vector Predicate. $\mathbf{Vp}(x)$ for "x is a vector."
- Zero Vector. $\vec{0}$
- Vector Addition. $\mathbf{v}_1 + \mathbf{v}_2$
- Vector Minus. $-\mathbf{v}$
- Scalar Multiplication. For real number a, $a \circ \mathbf{v}$
- Vector Multipication. $\mathbf{v}_1 \bullet \mathbf{v}_2$
- Unit Vector. $\vec{1}$
- Vector Norm. $\|\mathbf{v}\|$
- Vector Dot Product. $\mathbf{v}_1 \odot \mathbf{v}_2$
- Predicate with Quantifier. $\forall u[\mathbf{Vp}(u) \rightarrow u \odot x = 0]$
- Skolem Function. Witness function for Predicate with Quantifier

In addition to the usual closure axioms, the `encapsulate` adds these axioms to ACL2(r):

- Real Vector Space Axioms.

$$[\mathbf{Vp}(x) \wedge \mathbf{Vp}(y) \wedge \mathbf{Vp}(z)] \rightarrow (x+y)+z = x+(y+z)$$

$$[\mathbf{Vp}(x) \wedge \mathbf{Vp}(y)] \rightarrow x+y = y+x$$

$$\mathbf{Vp}(x) \rightarrow \vec{0}+x = x$$

$$\mathbf{Vp}(x) \rightarrow x+(-x) = \vec{0}$$

$$[\mathtt{Realp}(a) \wedge \mathtt{Realp}(b) \wedge \mathbf{Vp}(x)] \rightarrow a \circ (b \circ x) = (a \cdot b) \circ x$$

$$\mathbf{Vp}(x) \rightarrow 1 \circ x = x$$

$$[\mathtt{Realp}(a) \wedge \mathtt{Realp}(b) \wedge \mathbf{Vp}(x)] \rightarrow (a+b) \circ x = (a \circ x) + (b \circ x)$$

$$[\mathtt{Realp}(a) \wedge \mathbf{Vp}(x) \wedge \mathbf{Vp}(y)] \rightarrow a \circ (x+y) = (a \circ x) + (a \circ y)$$

- Real Vector Algebra Axioms.

$$\vec{1} \neq \vec{0}$$

$$[\texttt{Realp}(a) \wedge \texttt{Realp}(b) \wedge \mathbf{Vp}(x) \wedge \mathbf{Vp}(y) \wedge \mathbf{Vp}(z)] \rightarrow \\ x \bullet [(a \circ y) + (b \circ z)] = [a \circ (x \bullet y)] + [b \circ (x \bullet z)]$$

$$[\texttt{Realp}(a) \wedge \texttt{Realp}(b) \wedge \mathbf{Vp}(x) \wedge \mathbf{Vp}(y) \wedge \mathbf{Vp}(z)] \rightarrow \\ [(a \circ x) + (b \circ y)] \bullet z = [a \circ (x \bullet z)] + [b \circ (y \bullet z)]$$

$$\mathbf{Vp}(x) \rightarrow [(\vec{1} \bullet x = x) \wedge (x \bullet \vec{1} = x)]$$

- Vector Norm and Dot Product Axioms.

$$\mathbf{Vp}(x) \rightarrow [\texttt{Realp}(\|x\|) \wedge \|x\| \geq 0]$$

$$\mathbf{Vp}(x) \rightarrow [(\|x\| = 0) = (x = \vec{0})]$$

$$[\mathbf{Vp}(x) \wedge \mathbf{Vp}(y)] \rightarrow \|x \bullet y\| = \|x\| \cdot \|y\|$$

$$x \odot y = \frac{1}{2} \cdot [\|x+y\| - \|x\| - \|y\|]$$

$$[\texttt{Realp}(a) \wedge \texttt{Realp}(b) \wedge \mathbf{Vp}(x) \wedge \mathbf{Vp}(y) \wedge \mathbf{Vp}(z)] \rightarrow \\ [(a \circ x) + (b \circ y)] \odot z = [a \cdot (x \odot z)] + [b \cdot (y \odot z)]$$

$$\forall u[\mathbf{Vp}(u) \rightarrow u \odot x = 0] = \\ [\texttt{let } u \texttt{ be witness}(x)][\mathbf{Vp}(u) \rightarrow u \odot x = 0]$$

$$\forall u[\mathbf{Vp}(u) \rightarrow u \odot x = 0] \rightarrow [\mathbf{Vp}(u) \rightarrow u \odot x = 0]$$

$$(\mathbf{Vp}(x) \wedge \forall u[\mathbf{Vp}(u) \rightarrow u \odot x = 0]) \rightarrow x = \vec{0}$$

The ACL2(r) theory of composition algebras includes the following theorems and definitions:

- Scaling Laws.

$$[\mathbf{Vp}(x) \wedge \mathbf{Vp}(y) \wedge \mathbf{Vp}(z)] \rightarrow (x \bullet y) \odot (x \bullet z) = \|x\| \cdot (y \odot z)$$

$$[\mathbf{Vp}(x) \wedge \mathbf{Vp}(y) \wedge \mathbf{Vp}(z)] \rightarrow (x \bullet z) \odot (y \bullet z) = (x \odot y) \cdot \|z\|$$

- Exchange Law.

$$[\mathbf{Vp}(u) \wedge \mathbf{Vp}(x) \wedge \mathbf{Vp}(y) \wedge \mathbf{Vp}(z)] \rightarrow \\ [(u \bullet y) \odot (x \bullet z)] + [(u \bullet z) \odot (x \bullet y)] = \\ 2 \cdot (u \odot x) \cdot (y \odot z)$$

- Conjugate Definition.

$$\bar{x} = ([2 \cdot (x \odot \vec{1})] \circ \vec{1}) + (-x)$$

- Conjugate Laws.

$$[\mathbf{Vp}(x) \wedge \mathbf{Vp}(y) \wedge \mathbf{Vp}(z)] \rightarrow y \odot (\bar{x} \bullet z) = z \odot (x \bullet y)$$

$$[\mathbf{Vp}(x) \wedge \mathbf{Vp}(y) \wedge \mathbf{Vp}(z)] \rightarrow x \odot (z \bullet \bar{y}) = z \odot (x \bullet y)$$

$$\mathbf{Vp}(x) \rightarrow \bar{\bar{x}} = x$$

$$[\mathbf{Vp}(x) \wedge \mathbf{Vp}(y)] \rightarrow \overline{x \bullet y} = \bar{y} \bullet \bar{x}$$

- Inverse Definition.

$$x^{-1} = \frac{1}{\|x\|} \circ \bar{x}$$

- Inverse Law.

$$[\mathbf{Vp}(x) \wedge x \neq \vec{0}] \rightarrow [x^{-1} \bullet x = \vec{1} \wedge x \bullet x^{-1} = \vec{1}]$$

- Alternative Laws. Special versions of associativity.

$$[\mathbf{Vp}(x) \wedge \mathbf{Vp}(y)] \rightarrow x \bullet (x \bullet y) = (x \bullet x) \bullet y$$

$$[\mathbf{Vp}(x) \wedge \mathbf{Vp}(y)] \rightarrow (y \bullet x) \bullet x = y \bullet (x \bullet x)$$

$$[\mathbf{Vp}(x) \wedge \mathbf{Vp}(y)] \rightarrow (x \bullet y) \bullet x = x \bullet (y \bullet x)$$

- Other Theorems.

$$[\mathbf{Vp}(x) \wedge \mathbf{Vp}(y)] \rightarrow ([x \bullet y = \vec{0}] = [(x = \vec{0}) \vee (y = \vec{0})])$$

$$(\mathbf{Vp}(x) \wedge \mathbf{Vp}(y) \wedge \forall u[\mathbf{Vp}(u) \rightarrow u \odot x = u \odot y]) \rightarrow x = y$$

$$\|x\| = x \odot x$$

3 Composition Algebra Doubling.

Use `encapsulate` to axiomatize, in ACL2(r), two composition algebras, with vector predicates $V_1 p$ and $V_2 p$. The vectors satisfying $V_2 p$ are `ordered pairs` of elements satisfying $V_1 p$. Both algebras, individually, satisfy all the axioms (and also all theorems) of the previous section. These additional axioms connect the various vector operations of the two spaces:

- Additional Axioms.

$$x +_2 y = ([\mathtt{car}(x) +_1 \mathtt{car}(y)]; [\mathtt{cdr}(x) +_1 \mathtt{cdr}(y)])$$

$$a \circ_2 x = ([a \circ_1 \mathtt{car}(x)]; [a \circ_1 \mathtt{cdr}(x)])$$

$$[V_1 p(x) \wedge V_1 p(y)] \rightarrow (x; \vec{0}_1) \bullet_2 (y; \vec{0}_1) = ([x \bullet_1 y]; \vec{0}_1)$$

$$V_1 p(x) \rightarrow ([\|(x; \vec{0}_1)\|_2 = \|x\|_1] \wedge [\|(\vec{0}_1; x)\|_2 = \|x\|_1])$$

$$[V_1 p(x) \wedge V_1 p(y)] \rightarrow (x; \vec{0}_1) \odot_2 (\vec{0}_1; y) = 0$$

$$\vec{1}_2 = (\vec{1}_1; \vec{0}_1)$$

$$V_1 p(x) \rightarrow (x; \vec{0}_1) \bullet_2 (\vec{0}_1; \vec{1}_1) = (\vec{0}_1; x)$$

Since both $V_1 p$ and $V_2 p$ are composition algebras,

$$\begin{array}{l} [V_1 p(v_1) \wedge V_1 p(v_2) \wedge V_1 p(w_1) \wedge V_1 p(w_2)] \rightarrow \\ \quad [\, \|v_1 \bullet_1 v_2\|_1 = \|v_1\|_1 \cdot \|v_2\|_1 \wedge \\ \quad \|(v_1; w_1) \bullet_2 (v_2; w_2)\|_2 = \|(v_1; w_1)\|_2 \cdot \|(v_2; w_2)\|_2 \,]. \end{array}$$

Among the consequences of these `encapsulated` axioms, ACL2(r) verifies:

- Dot Product Doubling.

$$\begin{array}{l} [V_1 p(v_1) \wedge V_1 p(v_2) \wedge V_1 p(w_1) \wedge V_1 p(w_2)] \rightarrow \\ \quad (v_1; v_2) \odot_2 (w_1; w_2) = [v_1 \odot_1 w_1] + [v_2 \odot_1 w_2] \end{array}$$

- Conjugation Doubling.

$$[V_1 p(v_1) \wedge V_1 p(v_2)] \rightarrow \overline{(v_1; v_2)} = (\bar{v}_1; -v_2)$$

- Product Doubling.

$$\begin{array}{l} [V_1 p(v_1) \wedge V_1 p(v_2) \wedge V_1 p(w_1) \wedge V_1 p(w_2)] \rightarrow \\ \quad (v_1; v_2) \bullet_2 (w_1; w_2) = \\ \quad (\{[v_1 \bullet_1 w_1] - [\bar{w}_2 \bullet_1 v_2]\}; \{[w_2 \bullet_1 v_1] +_1 [v_2 \bullet_1 \bar{w}_1]\}) \end{array}$$

- Norm Doubling.
$$[V_1p(v_1) \wedge V_1p(v_2)] \rightarrow \|(v_1;v_2)\|_2 = \|v_1\|_1 + \|v_2\|_1$$
- Associativity of $\bullet_1$.
$$[V_1p(v_1) \wedge V_1p(v_2) \wedge V_1p(v_3)] \rightarrow [v_1 \bullet_1 v_2] \bullet_1 v_3 = v_1 \bullet_1 [v_2 \bullet_1 v_3]$$

The above doubling theorems match the definitions used in the Cayley-Dickson Construction for making ordered pairs of V_1p vectors into V_2p vectors. Furthermore, if both the V_2p ordered pairs and the the component V_1p vectors form composition algebras, then the component V_1p algebra has an associative multiplication.

In addition, ACL2(r) verifies that if the component V_1p vectors form a composition algebra with an associative multiplication, then the Cayley-Dickson Construction makes the V_2p ordered pairs into a composition algebra.

3.1 Summary

ACL2(r) verifies:

- Start with a composition algebra V_1p.
- Let V_2p be the set of ordered pairs of elements from V_1p.
- Then
  1. (a) If V_2p is also a composition algebra, then V_1p-multiplication is associative.
     (b) If V_1p-multiplication is associative, then V_2p can be made into a composition algebra.
  2. (a) If V_2p is a composition algebra with associative multiplication, then V_1p-multiplication is associative and commutative.
     (b) If V_1p-multiplication is associative and commutative, then V_2p can be made into a composition algebra with associative multiplication.
  3. (a) If V_2p is a composition algebra with associative and commutative multiplication, then V_1p-multiplication is also associative and commutative, and V_1p-conjugation is trivial.
     (b) If V_1p-multiplication is associative and commutative, and V_1p-conjugation is trivial, then V_2p can be made into a composition algebra with associative and commutative multiplication.

3.1.1 A last example

Apply the Cayley-Dickson Construction to pairs of octonions. Think of the octonions as eight-dimensional vectors. Interpret ordered pairs of octonions as sixteen-dimensional vectors called Sedenions [5]: For octonians

$$\begin{aligned} o &= a_1 + a_2 \cdot i + a_3 \cdot j + a_4 \cdot k + a_5 \cdot \ell + a_6 \cdot I + a_7 \cdot J + a_8 \cdot K \\ p &= b_1 + b_2 \cdot i + b_3 \cdot j + b_4 \cdot k + b_5 \cdot \ell + b_6 \cdot I + b_7 \cdot J + b_8 \cdot K, \end{aligned}$$

$$\begin{aligned} (o;p) &= o + p \cdot L \\ &= a_1 + a_2 \cdot i + a_3 \cdot j + a_4 \cdot k + a_5 \cdot \ell + a_6 \cdot I + a_7 \cdot J + a_8 \cdot K \\ &\quad + b_1 \cdot L + b_2 \cdot iL + b_3 \cdot jL + b_4 \cdot kL + b_5 \cdot \ell L + b_6 \cdot IL \\ &\quad + b_7 \cdot JL + b_8 \cdot KL. \end{aligned}$$

For octonion

$$\begin{aligned} o &= a_1 + a_2 \cdot i + a_3 \cdot j + a_4 \cdot k + a_5 \cdot \ell + a_6 \cdot I + a_7 \cdot J + a_8 \cdot K, \\ \bar{o} &= a_1 - a_2 \cdot i - a_3 \cdot j - a_4 \cdot k - a_5 \cdot \ell - a_6 \cdot I - a_7 \cdot J - a_8 \cdot K \end{aligned}$$

is the **conjugate** of o.

For octonions o_1, o_2, p_1, and p_2, sedenion multiplication is defined by

$$\begin{aligned} (o_1; o_2) \cdot (p_1; p_2) &= (o_1 + o_2 \cdot L) \cdot (p_1 + p_2 \cdot L) \\ &= (o_1 p_1 - \bar{p}_2 o_2) + (p_2 o_1 + o_2 \bar{p}_1) \cdot L \\ &= ([o_1 p_1 - \bar{p}_2 o_2]; [p_2 o_1 + o_2 \bar{p}_1]) \end{aligned}$$

Recall the octonians have a **non**-trivial conjugate and octonian multiplication is **not** commutative and also **not** associative. The octonions form a composition algebra, so that for octonions o and p, $\|o \cdot p\| = \|o\| \cdot \|p\|$.

By item 1(a) listed above about composition algebras, since octonian multiplication is not associative, the sedenions is not a composition algebra. In fact, the sedenion product of nonzero vectors could be the zero vector. For example, let o_1, o_2, p_1, and p_2 be these otonions:

$$\begin{aligned} o_1 &= 0 + 0 \cdot i + 0 \cdot j + 1 \cdot k + 0 \cdot \ell + 0 \cdot I + 0 \cdot J + 0 \cdot K \\ o_2 &= 0 + 0 \cdot i + 1 \cdot j + 0 \cdot k + 0 \cdot \ell + 0 \cdot I + 0 \cdot J + 0 \cdot K \\ p_1 &= 0 + 0 \cdot i + 0 \cdot j + 0 \cdot k + 0 \cdot \ell + 0 \cdot I + 1 \cdot J + 0 \cdot K \\ p_2 &= 0 + 0 \cdot i + 0 \cdot j + 0 \cdot k + 0 \cdot \ell + 0 \cdot I + 0 \cdot J + (-1 \cdot K). \end{aligned}$$

Then

$$\begin{aligned} &(o_1 + o_2 \cdot L) \cdot (p_1 + p_2 \cdot L) = \\ &\quad (0 + 0 \cdot i + 0 \cdot j + 0 \cdot k + 0 \cdot \ell + 0 \cdot I + 0 \cdot J + 0 \cdot K) \\ &\quad + (0 + 0 \cdot i + 0 \cdot j + 0 \cdot k + 0 \cdot \ell + 0 \cdot I + 0 \cdot J + 0 \cdot K) \cdot L. \end{aligned}$$

So

$$\|(o_1 + o_2 \cdot L) \cdot (p_1 + p_2 \cdot L)\| = 0,$$

but

$$\|(o_1 + o_2 \cdot L)\| = 2 = \|(p_1 + p_2 \cdot L)\|,$$

and

$$\|(o_1 + o_2 \cdot L) \cdot (p_1 + p_2 \cdot L)\| \neq \|(o_1 + o_2 \cdot L)\| \cdot \|(p_1 + p_2 \cdot L)\|.$$

All nonzero sedenions have multiplicative inverses. For example,

$$\begin{aligned} &(o_1 + o_2 \cdot L)^{-1} = \\ &\quad (0 + 0 \cdot i + 0 \cdot j + (-\frac{1}{2} \cdot k) + 0 \cdot \ell + 0 \cdot I + 0 \cdot J + 0 \cdot K) \\ &\quad + (0 + 0 \cdot i + (-\frac{1}{2} \cdot j) + 0 \cdot k + 0 \cdot \ell + 0 \cdot I + 0 \cdot J + 0 \cdot K) \cdot L \\ &(p_1 + p_2 \cdot L)^{-1} = \\ &\quad (0 + 0 \cdot i + 0 \cdot j + 0 \cdot k) + 0 \cdot \ell + 0 \cdot I + (-\frac{1}{2} \cdot J) + 0 \cdot K) \\ &\quad + (0 + 0 \cdot i + 0 \cdot j + 0 \cdot k + 0 \cdot \ell + 0 \cdot I + 0 \cdot J + (\frac{1}{2} \cdot K)) \cdot L \end{aligned}$$

A ACL2(r) Books

A.1 cayley1.lisp

The Reals form a (1-dimensional) composition algebra.

A.2 cayley1a.lisp

Cons pairs of Reals form a (2-dimensional) composition algebra.
This algebra is (isomorphic to) the Complex Numbers.

A.3 cayley1b.lisp

Cons pairs of Complex Numbers form a (4-dimensional) composition algebra.
This algebra is (isomorphic to) the Quaternions.

3-Dimensional Vector Cross Product and 3-Dimensional Dot Product are related to 4-Dimensional Quaternion Multiplication.

A.4 cayley1c.lisp

Cons pairs of Quaternions form a (8-dimensional) composition algebra.
This algebra is (isomorphic to) the Octonions.

7-Dimensional Vector Cross Product and 7-Dimensional Dot Product are related to 8-Dimensional Octonion Multiplication.

A.5 cayley1d.lisp

Cons pairs of Octonions form a (16-dimensional) algebra.
This algebra is (isomorphic to) the Sedenions.

This algebra is not a composition algebra, but all nonzero Sedenions have multiplicative inverses.

A.6 cayley2.lisp

Axioms and theory of composition algebras.

A.7 cayley2.lisp

In composition algebras, $\|v\| = v \odot v$.

A.8 cayley3.lisp

Start with a composition algebra V1. Let V2 be the set of ordered pairs of elements from V1.

If V2 is also a composition algebra, then V1-multiplication is associative.

A.9 cayley3a.lisp

Start with a composition algebra V. Let V2 be the set of ordered pairs of elements from V.

If V-multiplication is associative, then V2 can be made into a composition algebra.

A.10 `cayley4.lisp`

Start with a composition algebra V1. Let V2 be the set of ordered pairs of elements from V1.

If V2 is a composition algebra with associative multiplication, then V1-multiplication is associative and commutative.

A.11 `cayley4a.lisp`

Start with a composition algebra V. Let V2 be the set of ordered pairs of elements from V.

If V-multiplication is associative and commutative, then V2 can be made into a composition algebra with associative multiplication.

A.12 `cayley5.lisp`

Start with a composition algebra V1. Let V2 be the set of ordered pairs of elements from V1.

If V2 is a composition algebra with associative and commutative multiplication, then V1-multiplication is associative and commutative, and V1-conjugation is trivial.

A.13 `cayley5a.lisp`

Start with a composition algebra V. Let V2 be the set of ordered pairs of elements from V.

If V-multiplication is associative and commutative, and V-conjugation is trivial, then V2 can be made into a composition algebra with associative and commutative multiplication.

References

[1] J.H. Conway & D.A. Smith (2003): *On Quaternions and Octonions: Their Geometry, Arithmetic, and Symmetry*. AK Peters.

[2] Princeton Companion Group (2008): *Quaternions, Octonians, and Normed Division Algebras*. In Timothy Gowers, editor: *The Princeton Companion to Mathematics*, Princeton University Press, doi:10.1515/9781400830398.

[3] M. Koecher & R. Remmert (1991): *Chapter 7, Hamilton's Quaternions; Chapter 9, CAYLEY Numbers or Alternative Division Algebras; Chapter 10, Section 1, Composition Algebras*. In F. Hirzebruch M. Koecher K. Mainzer J. Neukirch A. Prestel H.-D. Ebbinghaus, H. Hermes & R. Remmert, editors: *Numbers*, Springer-Verlag, doi:10.1007/978-1-4612-1005-4.

[4] Wikipedia (2016): *Cayley-Dickson construction: From Wikipedia, the free encyclopedia*. Available at `https://en.wikipedia.org/wiki/Cayley-Dickson_construction`.

[5] Wikipedia (2016): *Sedenion: From Wikipedia, the free encyclopedia*. Available at `https://en.wikipedia.org/wiki/Sedenion`.

A Computationally Surveyable Proof of the Group Properties of an Elliptic Curve

David M. Russinoff
david@russinoff.com

We present an elementary proof of the abelian group properties of the elliptic curve known as *Curve25519*, as a component of a comprehensive proof of correctness of a hardware implementation of the associated Diffie-Hellman key agreement algorithm. The entire proof has been formalized and mechanically verified with ACL2, and is *computationally surveyable* in the sense that all steps that require mechanical support are presented in such a way that they may be readily reproduced in any suitable programming language.

1 Introduction

An effort is under way at Intel to develop and verify a formal model and a hardware implementation of the elliptic curve key agreement algorithm known as *Curve25519* [2], using ACL2. The most challenging aspect of this problem is the proof of the abelian group properties of the curve, especially associativity. This result may be viewed either as a deep theorem of algebraic or projective geometry [6, 3], accessible only to experts in that field, or as an elementary but computationally intensive arithmetic exercise, involving, as Bernstein [2] observes, "standard (but lengthy) calculations". Silverman and Tate [10] attempt to quantify the effort with a "tongue-in-cheek estimate":

> Of course, there are a lot of cases to consider .... But in a few days you will be able to check associativity using these formulas. So we need say nothing more about the proof of the associative law!

What remains to be said is that there is compelling evidence (see below) that an elementary hand proof of this result is a practical impossibility. The first serious attack on the problem, by Friedl [4], was a combination of mathematical analysis and symbolic computation with the CoCoA (Computations in Commutative Algebra) package. Building on Friedl's results, Théry [11] later developed a comprehensive formal proof with Coq. (Both papers address the somewhat more general class of Weierstrass curves rather than the one on which we focus here, but there is no difference in computational complexity.) These two efforts, which together represent a significant achievement, may be contrasted in the terminology of automated reasoning [1]: Friedl's work is *accepting* insofar as it treats CoCoA as a trusted oracle, whereas Théry's proof is *autarkic* by virtue of performing all logical deductions and supporting computations within the same formal system.

From a traditional mathematical perspective, however, both of these results are open to the same common criticism of computer-assisted proofs. There is general agreement in the mathematical community that it is desirable for a proof to be *surveyable* in the sense that each of its assertions could be derived manually by a competent reader as a logical consequence of preceding assertions and otherwise established results, and that the proof is short enough to be comprehended. One goal behind this principle is correctness, but equally important is the desire for mathematical growth—the propagation and advancement of techniques and ideas.

Warren Hunt Jr. and Anna Slobodova (Eds.):
ACL2 Theorem Prover and its Applications
EPTCS 249, 2017, pp. 30–46, doi:10.4204/EPTCS.249.3

In the realm of computing, this is often an unattainable objective—reliance on a mechanical proof assistant may be unavoidable. It is common to find in a published proof in this field, in lieu of a cogent argument, an appeal to the authority of an established proof system. This device is a stark realization of Tymoczko's allegory of the infallible Martian genius whose proclamations go unquestioned—proof by "Simon Says" [12]. It may provide evidence of correctness but does little to illuminate the underlying mathematics.

Dependence on mechanical assistance, however, need not preclude a full exposition of a proof. For example, the correctness of a hardware divider typically depends on a relation between the value and indices of each entry of an array that is too large to be either generated or checked by hand, but it should be possible to characterize the computation in such a way that it can be understood and machine-checked by the skeptical reader.

This suggests a judicious weakening of the conventional notion of surveyability. A proof may be said to be *computationally surveyable* if its only departure from that notion is its dependence on unproved assertions that satisfy the following criteria:

(1) The assertion pertains to a function for which a clear constructive definition has been provided, and merely specifies the value of that function corresponding to a concrete set of arguments.

(2) The computation of this value has been performed mechanically by the author of the proof in a reasonably short time.

(3) A competent reader could readily code the function in the programming language of his choice and verify the asserted result on his own computing platform.

Such a proof, though still objectionable to those who insist on strict surveyability, can convey a comprehensive understanding of a theorem and is susceptible to a process of social review, thus oppugning a commonly stated basis for the objection.

Neither of the treatments of the elliptic curve group properties cited above attempts such a proof, perhaps because the supporting tool or its application to the problem at hand is too complicated to admit a concise specification. Thus, Friedl simply attributes unproved results to CoCoA, while Théry's claims depend on an undisclosed "tactic" that has reportedly been implemented in Coq.

An integrated computationally surveyable proof of a result of this sort, which combines subtle mathematical analysis with intensive computation, is best carried out with the support of an interactive prover based on an efficient executable logic. We shall present such a proof of this theorem that has been formalized and mechanically checked in its entirety with ACL2. The computational results for which we rely solely on ACL2 for verification, as opposed to proof checking, (all of which are in Section 7) are labeled as **Computation**s, and are thus clearly distinguished from other steps in the proof, which are listed as **Lemma**s. All computations are performed on S-expressions and are most naturally performed in LISP, but can be readily implemented in any language that provides linked lists. Moreover, our exposition is confined to conventional mathematical terminology and notation, with no reference to the ACL2 logic.

Our proof benefits significantly from the two earlier efforts, both in its overall approach and through its appropriation of specific lemmas. In particular, we follow [11] in the representation of polynomials in sparse Horner normal form, using a normalization procedure adapted from [5]. Furthermore, our Lemmas 2.1, 2.2, and 7.7 are variants of results found in [11] (two of which are inherited from [4]).

The supporting materials for this paper include several subdirectories of `books/projects` in the ACL2 repository. The main script resides in `curve25519`. The basis of Curve25519 is the primality of $\wp = 2^{255} - 19$, which is proved in `quadratic-reciprocity` by Pratt's method [7]and explained in [9]. Fermat's Theorem ($a^{\wp-1} \bmod \wp = 1$ when a is not divisible by $\wp$), which allows the inversion operator in the field $\mathbb{F}_\wp$ to be defined as $a^{-1} = a^{\wp-2} \bmod \wp$, is also formalized in `quadratic-reciprocity`.

Our formalization of sparse Horner normal forms is in the subdirectory `shnf`. Following [5], we define an efficiently computable normalization of polynomial terms and an evaluation function on normal forms, and prove equality between the value of a polynomial and that of its representation, for all variable assignments. Thus, the equivalence of two polynomials we may be established by computing their normalizations and observing that they coincide.

Of course, the utility of this method rests on the property of completeness: equivalent polynomials always produce the same representation. According to the authors of the Coq proof, which does not address this property, it cannot even be stated within their formal framework. Our development includes a constructive proof of this result that we have formalized in ACL2, based on a function that computes, for a given pair of two polynomials, a list of variable assignments for which the values of the polynomials differ, whenever such a list exists. This result is not required for our present purpose, but is documented elsewhere [8].

The distinguishing features of our proof that enable the objective of computational surveyability are (1) a specialized rewriting procedure that reduces the normal form of a polynomial according to the curve equation (Definition 5.5), and (2) an encoding of group elements as integer triples, which facilitates symbolic computation of the group operation (Definition 6.3). Both of these functions require automated computation but admit concise specifications and correctness proofs. Furthermore, a modest improvement in efficiency over the more general Coq proof tactic is suggested by a comparison of execution times of the three computational results of [11] (9.2, 3.9, and 18.8 seconds for `spec3_assoc`, `spec2_assoc`, and `spec3_assoc`, respectively) and our versions of the same computations (3.78, 0.36, and 3.8 seconds for Computations 7, 8, and 9). We exploit this facility by performing several more intensive computations, thereby eliminating much of Théry's analysis, which he characterizes as "really tedious". In particular, Computation 10, which is proved in 26.2 seconds, disposes of a critical case of associativity. It is also worth noting that if the polynomial involved in this result were expanded into a sum of monomials, as might be done in a direct hand proof based on "standard computations", the number of terms would exceed 10^{25}. Clearly, the reader who completes such a proof "in a few days" is exceptionally good with figures.

2 Curve25519

Let $\wp = 2^{255} - 19$ and $A = 486662$. The primality of $\wp$ is proved in [9]. The field of order $\wp$ is the set $\mathbb{F}_\wp = \{0, 1, 2, \ldots, \wp - 1\}$ with the operations of addition and multiplication modulo $\wp$. Every $n \in \mathbb{Z}$ naturally corresponds to the field element n mod $\wp$, which we denote as $\bar{n}$. The field operations will be denoted by the usual symbols: if $x \in \mathbb{F}_\wp$, $y \in \mathbb{F}_\wp$, and $k \in \mathbb{N}$, then "$x+y$", "$x-y$", "$-x$", "xy", or "x^k" may refer to an operation in either $\mathbb{F}_\wp$ or $\mathbb{Z}$, depending on context, whereas "x/y" will only denote an operation in $\mathbb{F}_\wp$.

Definition 2.1 $EC = \{(x,y) \in \mathbb{F}_\wp \times \mathbb{F}_\wp \mid y^2 = x^3 + Ax^2 + x\} \cup \{\infty\}$.

Our goal is to show that EC is an abelian group under the following operation:

Definition 2.2 *Let $P \in EC$ and $Q \in EC$.*

(1) $P \oplus \infty = \infty \oplus P = P$.

(2) If $P = (x,y)$, then $P \oplus (x,-y) = \infty$.

(3) If $P = (x_1,y_1)$, $Q = (x_2,y_2) \neq (x_1,-y_1)$, and $\lambda = \begin{cases} \frac{y_2-y_1}{x_2-x_1} & \text{if } x_1 \neq x_2 \\ \frac{3x_1^2+2Ax_1+1}{2y_1} & \text{if } x_1 = x_2, \end{cases}$ *then $P \oplus Q = (x,y)$, where $x = \lambda^2 - A - x_1 - x_2$ and $y = \lambda(x_1 - x) - y_1$.*

Clearly, ∞ is the identity element, the inverse of $P = (x,y)$ is $\ominus P = (x,-y)$, and according to Corollary 1 of [9], the origin $O = (0,0)$ is the only element of order 2.

Remark. If we consider Definition 2.1 as an equation over $\mathbb{R}$ instead of $\mathbb{F}_{\wp}$, then Definition 2.2 admits a simple geometric interpretation. Except when $Q = \ominus P$, the line connecting points P and Q on the curve (or the tangent line at P, in case $P = Q$) intersects the curve at another point, R. If we were to define the operation as $P \oplus Q = \ominus R$, then analytic geometry would yield the formula in the definition. If $Q = \ominus P$, the third point of intersection is taken to be ∞.

In the sequel, we shall assume that $P_0 = (x_0, y_0)$, $P_1 = (x_1, y_1)$, and $P_2 = (x_2, y_2)$ are fixed elements of EC that are distinct from ∞ (but not necessarily from one another), in order to obviate repetition of such hypotheses. Any result pertaining to these points may be generalized by replacing them with arbitrary finite points of EC. We begin with two simple consequences of Definition 2.2.

Lemma 2.1 $P_0 \oplus P_1 \neq P_0$.

PROOF: Suppose $P_0 \oplus P_1 = P_0$. Equating y-coordinates, we have $y_0 = \lambda(x_0 - x_0) - y_0 = -y_0$, which implies $2y_0 = 0$ and hence (since $\mathbb{F}_{\wp}$ is of odd characteristic $\wp$) $y_0 = 0$, which implies $x_0 = 0$. But x_1 cannot be 0, as this would imply $P_0 = O \oplus O = \infty$. Thus, the equation $x_0 = \lambda^2 - A - x_0 - x_1$ reduces to

$$\frac{y_1^2}{x_1^2} = \lambda^2 = x_1 + A,$$

which implies $x_1^3 + Ax_1^2 + x_1 = y_1^2 = x_1^3 + Ax_1^2$, contradicting $x_1 \neq 0$. □

Lemma 2.2 *If $P_0 \oplus P_1 = P_0 \oplus (\ominus P_1)$, then either $P_0 = O$ or $P_1 = O$.*

PROOF: If $P_0 = \ominus P_1$, then

$$P_0 \oplus P_0 = P_0 \oplus (\ominus P_1) = P_0 \oplus P_1 = \ominus P_1 \oplus P_1 = \infty,$$

which implies $P_0 = O$. Similarly, if $P_0 = P_1$, then

$$P_0 \oplus P_0 = P_0 \oplus P_1 = P_0 \oplus (\ominus P_1) = P_0 \oplus (\ominus P_0) = \infty.$$

Therefore, we may assume $x_0 \neq x_1$. Equating the x-coordinates of $P_0 \oplus P_1$ and $P_0 \oplus (\ominus P_1)$, we have

$$\left(\frac{y_1 - y_0}{x_1 - x_0}\right)^2 - x_0 - x_1 = \left(\frac{y_1 + y_0}{x_1 - x_0}\right)^2 - x_0 - x_1,$$

which implies $4y_0y_1 = 0$, and hence either $y_0 = 0$ or $y_1 = 0$. □

3 Encoding Points on the Curve as Integer Triples

Our scheme for symbolic computation of the group operation is based on a mapping from $\mathbb{Z}^3$ to $\mathbb{F}_{\wp}^2$:

Definition 3.1 *If $\mathscr{P} = (m,n,z) \in \mathbb{Z}^3$, where z is not divisible by $\wp$, then*

$$decode(\mathscr{P}) = \left(\frac{\bar{m}}{\bar{z}^2}, \frac{\bar{n}}{\bar{z}^3}\right) \in \mathbb{F}_{\wp}^2$$

and $\mathscr{P}$ is said to be an encoding of $decode(\mathscr{P})$.

Note that every $P=(x,y)\in\mathbb{F}_{\wp}^2$ admits the canonical encoding $\mathscr{P}=(x,y,1)$.

The motivation for this definition is that an encoding of $P\oplus Q$ can often be readily derived from encodings of P and Q in certain cases of interest. We define a partial addition operation on $\mathbb{Z}^3$ corresponding to Definition 2.2:

Definition 3.2 *Given $\mathscr{P}\in\mathbb{Z}^3$ and $\mathscr{Q}\in\mathbb{Z}^3$, $\mathscr{P}\oplus\mathscr{Q}\in\mathbb{Z}^3$ is defined in two cases:*

(1) If $\mathscr{P}=\mathscr{Q}=(m,n,z)$, then $\mathscr{P}\oplus\mathscr{Q}=(m',n',z')$, where

$$\begin{aligned} z' &= z_{dbl}(\mathscr{P})=2nz,\\ w' &= w_{dbl}(\mathscr{P})=3m^2+2Amz^2+z^4,\\ m' &= m_{dbl}(\mathscr{P})=w'^2-4n^2(Az^2+2m),\\ n' &= n_{dbl}(\mathscr{P})=w'(4mn^2-m')-8n^4. \end{aligned}$$

(2) If $\mathscr{P}=(x,y,1)\in\mathbb{Z}^3$ and $\mathscr{Q}=(m,n,z)\neq\mathscr{P}$, then $\mathscr{P}\oplus\mathscr{Q}=(m',n',z')$, where

$$\begin{aligned} z' &= z_{sum}(\mathscr{P},\mathscr{Q})=z(z^2x-m),\\ m' &= m_{sum}(\mathscr{P},\mathscr{Q})=\left(z^3y-n\right)^2-\left(z^2(A+x)+m\right)\left(z^2x-m\right)^2\\ n' &= n_{sum}(\mathscr{P},\mathscr{Q})=\left(z^3y-n\right)\left(z'^2x-m'\right)-z'^3y. \end{aligned}$$

Lemma 3.1 *Let $P=decode(\mathscr{P})\in EC$ and $Q=decode(\mathscr{Q})\in EC$, where $\mathscr{P}\oplus\mathscr{Q}$ is defined and if $P=Q$, then $P\neq O$ and $\mathscr{P}=\mathscr{Q}$. Then $decode(\mathscr{P}\oplus\mathscr{Q})=P\oplus Q$.*

PROOF: The arithmetic operations below are to be understood as operations in $\mathbb{F}_{\wp}$ on the field elements corresponding to the integers involved.

We first consider the case $\mathscr{P}=\mathscr{Q}=(m,n,z)$. Let

$$\lambda=\frac{3\left(\frac{m}{z^2}\right)^2+2A\left(\frac{m}{z^2}\right)+1}{2\left(\frac{n}{z^3}\right)}=\frac{3m^2+2Amz^2+z^4}{2nz}=\frac{w'}{z'}.$$

Then $P\oplus P=(x,y)$, where

$$x=\lambda^2-A-2\left(\frac{m}{z^2}\right)=\frac{w'^2}{z'^2}-\frac{Az^2+2m}{z^2}=\frac{w'^2}{z'^2}-\frac{4n^2(Az^2+2m)}{z'^2}=\frac{m'}{z'^2}$$

and

$$y=\lambda\left(\frac{m}{z^2}-x\right)-\frac{n}{z^3}=\frac{w'}{z'}\cdot\frac{4mn^2-m'}{z'^2}-\frac{8n^4}{z'^3}=\frac{w'(4mn^2-m')-8n^4}{z'^3}=\frac{n'}{z'^3}.$$

Thus, $decode(\mathscr{P}\oplus\mathscr{P})=decode(m',n',z')=(x,y)=P\oplus P$.

In the remaining case, we have $\mathscr{P}=(m,n,z)$ and $\mathscr{Q}=(x,y,1)$. Let

$$\lambda=\frac{y-\frac{n}{z^3}}{x-\frac{m}{z^2}}=\frac{z^3y-n}{z(z^2x-m)}=\frac{z^3y-n}{z'},$$

Then $P\oplus Q=(x',y')$, where

$$\begin{aligned} x' &= \lambda^2-A-x-\left(\frac{m}{z^2}\right)=\frac{(z^3y-x)^2}{z'^2}-\frac{z^2(A+x)+m}{z^2}\\ &= \frac{(z^3y-x)^2}{z'^2}-\frac{\left(z^2(A+x)+m\right)\left(z^2x-m\right)^2}{z'^2}=\frac{m'}{z'^2} \end{aligned}$$

and

$$y' = \lambda(x - x') - y = \frac{z^3 y - n}{z'} \cdot \frac{xz'^2 - m'}{z'^2} - \frac{z'^3 y}{z'^3} = \frac{n'}{z'^3}.$$

Thus, $decode(\mathscr{P} \oplus \mathscr{Q}) = decode(m', n', z') = (x', y') = P \oplus Q$. □

4 Polynomial Terms and Sparse Horner Normal Form

In this section, we describe our formalization of sparse Horner forms as S-expressions. For this purpose, an S-expression is an integer, a symbol, or an ordered list $s = (s_0\ s_1\ \ldots\ s_n)$ of S-expressions. In the last case, $head(s) = s_0$, and for $k \in \mathbb{N}$, we define $s^{(k)} = (s_k\ \ldots\ s_n)$. The set of all lists whose members are confined to a set S is $\mathscr{L}(S)$.

Under the usual ACL2 encoding of multi-variable polynomials, a *polynomial term* over a list V of variable symbols is an S-expression constructed from integers and symbols in V using the symbols `+`, `-`, `*`, and `EXPT`. The function *evalp* evaluates a term according to an alist that associates variables with integer values in the natural way. For example, if $V =$ `(X Y Z)` and $A =$ `((X 2) (Y 3) (Z 0))`, then $\tau =$ `(* X (EXPT (+ Y Z) 3))` is a term over V and $evalp(\tau, A) = 2 \cdot (3+0)^3 = 54$. The set of all polynomial terms over V is denoted $\mathscr{T}(V)$.

We shall represent polynomial terms as objects of the following type:

Definition 4.1 *A sparse Horner form (SHF) is any of the following:*

(a) An integer;

(b) A list `(POP i p)`*, where* $i \in \mathbb{N}$ *and* p *is a SHF*

(c) A list `(POW i p q)`*, where* $i \in \mathbb{N}$ *and* p *and* q *are SHFs.*

A SHF is normal if its components are normal and it is not either of the following:

(a) `(POP i p)`*, where* $i = 0$ *or* $p \in \mathbb{Z}$ *or* $p =$ `(POP j q)`*;*

(b) `(POW i p q)`*, where* $i = 0$ *or* $p =$ `(POW j r 0)`*.*

$\mathscr{H}$ *denotes the set of all normal SHFs, or SHNFs.*

The evaluation of a SHF with respect to a list of integers is defined as follows:

Definition 4.2 *Let* h *be a SHF and let* $N \in \mathscr{L}(Z)$*.*

(a) If $h \in \mathbb{Z}$*, then* $evalh(h, N) = h$*.*

(b) If $h =$ `(POP i p)`*, then then* $evalh(h, N) = evalh(p, N^{(i)})$*.*

(c) If $h =$ `(POW i p q)` *and* $head(N) = n$*, then* $evalh(h, N) = n^i evalh(p, N) + evalh(q, N^{(1)})$*.*

(d) If $h =$ `(POW i p q)` *and* $N =$ `()`*, then* $evalh(h, N) = 0$*.*

Our objective is to define, for a given variable list $V =$ `(`$v_0 \ldots v_k$`)`, a mapping $norm : \mathscr{T}(V) \to \mathscr{H}$ such that if $N =$ `(`$n_0 \ldots n_k$`)` and $A =$ `((`v_0 n_0`)...(`v_k n_k`))`, then

$$evalh(norm(x, V), N) = evalp(x, A).$$

Thus, if two polynomials produce the same normal form, then they are equivalent.

A possible top-down approach to the definition of $norm(f, V)$ is as follows:

(1) If f is an integer constant, then $norm(f,V) = f$.

(2) Suppose v_0 occurs in f. Find polynomials g and h such that $f = v_0^i \cdot g + h$, g is not divisible by v_0, and v_0 does not occur in h. If $p = norm(g,V)$ and $q = norm(h,V^{(1)})$, then

$$norm(f,V) = \texttt{(POW } i\ p\ q\texttt{)}.$$

(3) Suppose v_0 does not occur in f. Let v_i be the first variable in V that does occur in f. If $p = norm(f,V^{(i)})$, then

$$norm(f,V) = \texttt{(POP } i\ p\texttt{)}.$$

For example, consider the polynomial $4x^4y^2 + 3x^3 + 2z^4 + 5$ with variable ordering $\texttt{(}x\ y\ z\texttt{)}$. Rewriting the polynomial as

$$x^3(4xy^2 + 3) + (2z^4 + 5),$$

we find that the normalization is $\texttt{(POW 3 } p\ q\texttt{)}$, where $p = norm(4xy^2 + 3, \texttt{(}x\ y\ z\texttt{)})$ and $q = norm(2z^4 + 5, \texttt{(}y\ z\texttt{)})$. Continuing recursively, we arrive at the final result:

```
(POW 3 (POW 1 (POP 1 (POW 2 4 0)) 3)
       (POP 1 (POW 4 2 5))).
```

It may be instructive to check that the value of this SHF for the list of values $(1\ 2\ 3)$, for example, and the value of the represented polynomial for the corresponding alist, are both 207.

It is not difficult to see that a SHF generated by this procedure is indeed normal. Unfortunately, this approach is impractical because of the general difficulty of constructing the polynomials g and h in Case (2). Our preferred definition will provide a more efficient bottom-up procedure. We begin with the two basic normalizing functions *pop* and *pow*:

Definition 4.3 *Let $i \in \mathbb{N}$ and $p \in \mathscr{H}$.*

(a) If $i = 0$ or $p \in \mathbb{Z}$, then $pop(i,p) = p$.

(b) If $p = \texttt{(POP } j\ q\texttt{)}$, then $pop(i,p) = \texttt{(POP } i+j\ q\texttt{)}$.

(c) Otherwise, $pop(i,p) = \texttt{(POP } i\ p\texttt{)}$.

Definition 4.4 *Let $i \in \mathbb{N} - \{0\}$, $p \in \mathscr{H}$, and $q \in \mathscr{H}$.*

(a) If $p = 0$, then $pow(i,p,q) = pop(1,q)$.

(b) If $p = \texttt{(POW } j\ r\ 0\texttt{)}$, then $pow(i,p,q) = \texttt{(POW } i+j\ r\ q\texttt{)}$.

(c) Otherwise, $pow(i,p,q) = \texttt{(POW } i\ p\ q\texttt{)}$.

The following properties of these functions are immediate consequences of the definitions:

Lemma 4.1 *Let $i \in \mathbb{N}$, $p \in \mathscr{H}$, $q \in \mathscr{H}$, and $N \in \mathscr{L}(Z)$.*

(a) $pop(i,p) \in \mathscr{H}$ and $evalh(pop(i,p),N) = evalh(\texttt{(POP } i\ p\texttt{)},N)$.

(b) If $i \neq 0$, then $pow(i,p,q) \in \mathscr{H}$ and $evalh(pow(i,p,q),N) = evalh(\texttt{(POW } i\ p\ q\texttt{)},N)$.

We also define a ring structure on $\mathscr{H}$, Once we have computed the SHNFs for polynomials x and y, the ring operations "$\oplus$" and "$\otimes$" compute those for $\texttt{(+ } x\ y\texttt{)}$ and $\texttt{(* } x\ y\texttt{)}$.

Definition 4.5 *Let $x \in \mathscr{H}$ and $x \in \mathscr{H}$.*

(1) If $x \in \mathbb{Z}$, then

(a) $y \in \mathbb{Z} \Rightarrow x \oplus y = x + y$ *and* $x \otimes y = xy$.

(b) $y = \texttt{(POP } i\ p\texttt{)} \Rightarrow x \oplus y = \texttt{(POP } i\ x \oplus p\texttt{)}$ *and* $x \otimes y = pop(i, x \otimes p)$.

(c) $y = \texttt{(POW } i\ p\ q\texttt{)} \Rightarrow x \oplus y = \texttt{(POW } i\ p\ x \oplus q\texttt{)}$ *and* $x \otimes y = pow(i, x \otimes p, x \otimes q)$.

(2) *If* $y \in \mathbb{Z}$, *then* $x \oplus y = y \oplus x$ *and* $x \otimes y = y \otimes x$.

(3) *If* $x = \texttt{(POP } i\ p\texttt{)}$ *and* $y = \texttt{(POP } j\ q\texttt{)}$, *then*

(a) $i = j \Rightarrow x \oplus y = pop(i, p \oplus q)$ *and* $x \otimes y = pop(i, p \otimes q)$.

(b) $i > j \Rightarrow x \oplus y = pop(j, \texttt{(POP } i - j\ p\texttt{)} \oplus q)$ *and* $x \otimes y = pop(j, \texttt{(POP } i - j\ p\texttt{)} \otimes q)$.

(c) $i < j \Rightarrow x \oplus y = pop(i, \texttt{(POP } j - i\ q\texttt{)} \oplus p)$ *and* $x \otimes y = pop(i, \texttt{(POP } j - i\ q\texttt{)} \otimes p)$.

(4) *If* $x = \texttt{(POP } i\ p\texttt{)}$ *and* $y = \texttt{(POW } j\ q\ r\texttt{)}$, *then*

(a) $i = 1 \Rightarrow x \oplus y = \texttt{(POW } j\ q\ r \oplus p\texttt{)}$ *and* $x \otimes y = \texttt{(POW } j\ x \otimes q\ p \otimes r\texttt{)}$.

(b) $i > 1 \Rightarrow x \oplus y = \texttt{(POW } j\ q\ r \oplus \texttt{(POP } i - 1\ p\texttt{))}$ *and* $x \otimes y = \texttt{(POW } j\ x \otimes q\ \texttt{(POP } i - 1\ p\texttt{)} \otimes r\texttt{)}$.

(5) *If* $y = \texttt{(POP } i\ p\texttt{)}$ *and* $y = \texttt{(POW } j\ q\ r\texttt{)}$, *then* $x \oplus y = y \oplus x$ *and* $x \otimes y = y \otimes x$.

(6) *If* $x = \texttt{(POW } i\ p\ q\texttt{)}$ *and* $y = \texttt{(POW } j\ r\ s\texttt{)}$, *then*

(a) $i = j \Rightarrow x \oplus y = pow(i, p \oplus r, q \oplus s)$.

(b) $i > j \Rightarrow x \oplus y = pow(j, \texttt{(POW } i - j\ p\ 0\texttt{)} \oplus r, q \oplus s)$

(c) $i < j \Rightarrow x \oplus y = pow(i, \texttt{(POW } j - i\ q\ 0\texttt{)} \oplus p, s \oplus q)$.

(d) $x \otimes y = (pow(i + j, p \otimes r, q \otimes s) \oplus pow(i, p \otimes pop(1, s), 0)) \oplus pow(i, r \otimes pop(1, q), 0)$.

The definitions of negation and exponentiation are straightforward:

Definition 4.6 *Let* $x \in \mathscr{H}$.

(1) *If* $x \in \mathbb{Z}$, *then* $\ominus x = -x$.

(2) *If* $x = \texttt{(POP } i\ p\texttt{)}$, *then* $\ominus x = \texttt{(POP } i\ \ominus p\texttt{)}$.

(3) *If* $x = \texttt{(POW } i\ p\ q\texttt{)}$, *then* $\ominus x = \texttt{(POW } i\ \ominus p\ \ominus q\texttt{)}$.

Definition 4.7 *If* $x \in \mathscr{H}$ *and* $k \in \mathbb{N}$, *then*

$$x^k = \begin{cases} 1 & \text{if } k = 0 \\ x \otimes x^{k-1} & \text{if } k > 0. \end{cases}$$

The following properties are easily proved by induction:

Lemma 4.2 *Let* $x \in \mathscr{H}$, $y \in \mathscr{H}$, $N \in \mathscr{L}(Z)$, *and* $k \in \mathbb{N}$.

(a) $x \oplus y \in \mathscr{H}$ *and* $evalh(x \oplus y, N) = evalh(x, N) + evalh(y, N)$.

(b) $x \otimes y \in \mathscr{H}$ *and* $evalh(x \otimes y, N) = evalh(x, N) \cdot evalh(y, N)$.

(c) $\ominus x \in \mathscr{H}$ *and* $evalh(\ominus x, N) = -evalh(x, N)$.

(d) $x^k \in \mathscr{H}$ *and* $evalh(x^k, N) = evalh(x, N)^k$.

We can now define the normalization procedure:

Definition 4.8 *If* $x \in \mathscr{T}(V)$, *where* $V = \texttt{(}v_0 \ldots v_{k-1}\texttt{)}$ *is a list of distinct symbols, then*

(1) $x \in \mathbb{Z} \Rightarrow norm(x, V) = x$.

(2) $x = v_i$, $0 \leq i < k \Rightarrow norm(x, V) = pop(i, \texttt{(POW 1 1 0)})$.

(3) $x = \texttt{(- } y\texttt{)} \Rightarrow norm(x,V) = \ominus norm(y,V)$.

(4) $x = \texttt{(+ } y\ z\texttt{)} \Rightarrow norm(x,V) = norm(y,V) \oplus norm(z,V)$.

(5) $x = \texttt{(- } y\ z\texttt{)} \Rightarrow norm(x,V) = norm(y,V) \oplus (\ominus norm(z,V))$.

(6) $x = \texttt{(* } y\ z\texttt{)} \Rightarrow norm(x,V) = norm(y,V) \otimes norm(z,V)$.

(7) $x = \texttt{(EXPT } y\ k\texttt{)} \Rightarrow norm(x,V) = norm(y,V)^k$.

The reader may wish to check that the SHNF for the polynomial $-z+x^3(z+x-3y)$ with respect to the variable list $\texttt{(}x\ y\ z\texttt{)}$ is once again

```
(POW 3 (POW 1 (POP 1 (POW 2 4 0)) 3)
       (POP 1 (POW 4 2 5))).
```

Lemma 4.3 *Let $f \in \mathscr{T}(V)$, where $V = \texttt{(}v_0 \ldots v_{k-1}\texttt{)}$ is a list of distinct symbols. Let $N = \texttt{(}n_0 \ldots n_{\ell-1}\texttt{)}$ be a list of integers with $\ell \geq k$ and*

$$A = \texttt{((}v_0\ n_0\texttt{)} \ldots \texttt{(}v_{k-1}\ n_{k-1}\texttt{))},$$

Then $norm(f,V) \in \mathscr{H}$ and

$$evalh(norm(f,V),N) = evalp(f,A).$$

PROOF: The case $f = v_i$ follows from Definition 4.2 and Lemma 4.1; the other cases follow from Definition 4.2, induction, and Lemma 4.2. □

5 Polynomial Reduction

We shall focus on the case of a list of variables corresponding to the coordinates of the points P_0, P_1, and P_2, as characterized in Section 2. We define the following lists:

Definition 5.1 $\mathscr{V} = \texttt{(Y0 Y1 Y2 X0 X1 X2)}$, $\mathscr{N} = \texttt{(}y_0\ y_1\ y_2\ x_0\ x_1\ x_2\texttt{)}$, *and*

$$\mathscr{A} = \texttt{((Y0 }y_0\texttt{) (Y1 }y_1\texttt{) (Y2 }y_2\texttt{) (X0 }x_0\texttt{) (X1 }x_1\texttt{) (X2 }x_2\texttt{))}$$

We abbreviate $\mathscr{T}(\mathscr{V})$ as $\mathscr{T}$, and for $\tau \in \mathscr{T}$ we abbreviate $evalp(\tau,\mathscr{A})$ as $\hat{\tau}$.

The ordering of the variable list $\mathscr{V}$ is designed to maximize the efficiency of the rewriting procedure defined below. This procedure operates on a SHF that represents a polynomial with respect to $\mathscr{V}$, which is effectively reduced, using the curve equation as a rewrite rule, to a polynomial that (a) has the same value (modulo $\wp$) as the given polynomial under the variable assignments of $\mathscr{A}$ and (b) is at most linear in each of the variables $\texttt{Y}i$.

The core of the rewriter is the function *split*, which reduces and splits a polynomial term τ into a sum of two polynomials, of which one is independent of a given $\texttt{Y}j$ and the other is linear in $\texttt{Y}j$. More precisely, if $h = norm(\tau, \mathscr{V}^{(k)})$ and $0 \leq k \leq j \leq 3$, then $split(h,j,k)$ computes a pair of SHNFs that represent these polynomials.

The following SHNF is used in the reduction:

Definition 5.2 $\Theta = \texttt{(POP 3 (POW 1 (POW 1 (POW 1 1 }A\texttt{) 1) 0))}$.

Lemma 5.1 *If $j \in \{0,1,2\}$, then $evalh(\Theta, \mathscr{N}^{(j)}) = x_j^3 + Ax_j^2 + x_j \equiv y_j^2 \pmod{\wp}$.*

PROOF: This may be derived by expanding the definition of *evalh*. □

Definition 5.3 *Let $h \in \mathscr{H}$, $j \in \{0,1,2\}$, and $k \in \mathbb{N}$.*

(1) If $h \in \mathbb{Z}$ or $j < k$, then $split(h,j,k) = (h,0)$.

(2) If $j \geq k$, $h = $ `(POP i p)`*, and $(p_0,p_1) = split(p,j,k+i)$, then*

$$split(h,j,k) = (pop(i,p_0),pop(i,p_1)).$$

(3) Let $h = $ `(POW i p q)`*, $(p_0,p_1) = split(p,j,k)$, and $(q_0,q_1) = split(q,j,k+1)$.*

(a) If $j > k$, then

$$split(h,j,k) = (pow(i,p_0,q_0),pow(i,p_1,q_1));$$

(b) If $j = k$ and i is even, then

$$split(h,j,k) = \left((\Theta^{\frac{i}{2}} \otimes p_0) \oplus pop(1,q_0), (\Theta^{\frac{i}{2}} \otimes p_1) \oplus pop(1,q_1)\right);$$

(c) If $j = k$ and i is odd, then

$$split(h,j,k) = \left((\Theta^{\frac{i+1}{2}} \otimes p_1) \oplus pop(1,q_0), (\Theta^{\frac{i-1}{2}} \otimes p_0) \oplus pop(1,q_1)\right).$$

Lemma 5.2 *Let $(h_0,h_1) = split(h,j,k)$, where $h \in \mathscr{H}$, $j \in \{0,1,2\}$, and $k \in \mathbb{N}$. Then $evalh(h,\mathscr{N}^{(k)}) \equiv evalh(h_0,\mathscr{N}^{(k)}) + y_j \cdot evalh(h_1,\mathscr{N}^{(k)}) \pmod{\wp}$.*

PROOF: We may assume that $j \geq k$; otherwise the claim is trivial. The proof is by induction on the structure of h. The case $h = $ `(POP i p)` is straightforward:

$$\begin{aligned} evalh(h,\mathscr{N}^{(k)}) &= evalh(p,\mathscr{N}^{(k+i)}) \\ &\equiv evalh(p_0,\mathscr{N}^{(k+i)}) + y_j \cdot evalh(p_1,\mathscr{N}^{(k+i)}) \\ &= evalh(pop(i,p_0),\mathscr{N}^{(k)}) + y_j \cdot evalh(pop(i,p_1),\mathscr{N}^{(k)}) \\ &= evalh(h_0,\mathscr{N}^{(k)}) + y_j \cdot evalh(h_1,\mathscr{N}^{(k)}). \end{aligned}$$

Suppose $h = $ `(POW i p q)`. By the definition of *evalh*,

$$\begin{aligned} &evalh(h,\mathscr{N}^{(k)}) \\ &= y_k^i \cdot evalh(p,\mathscr{N}^{(k)}) + evalh(q,\mathscr{N}^{(k+1)}) \\ &\equiv y_k^i \left(evalh(p_0,\mathscr{N}^{(k)}) + y_j \cdot evalh(p_1,\mathscr{N}^{(k)})\right) + \left(evalh(q_0,\mathscr{N}^{(k+1)}) + y_j \cdot evalh(q_1,\mathscr{N}^{(k+1)})\right) \\ &= \left(y_k^i evalh(p_0,\mathscr{N}^{(k)}) + evalh(q_0,\mathscr{N}^{(k+1)})\right) + y_j \cdot \left(y_k^i evalh(p_1,\mathscr{N}^{(k)}) + evalh(q_1,\mathscr{N}^{(k+1)})\right). \end{aligned}$$

If $j > k$, then this may be written as

$$\begin{aligned} &evalh(\texttt{(POW i } p_0\ q_0\texttt{)},\mathscr{N}^{(k)}) + y_j \cdot evalh(\texttt{(POW i } p_1\ q_1\texttt{)},\mathscr{N}^{(k)}) \\ &= evalh(pow(i,p_0,q_0),\mathscr{N}^{(k)}) + y_j \cdot evalh(pow(i,p_1,q_1),\mathscr{N}^{(k)}) \\ &= evalh(h_0,\mathscr{N}^{(k)}) + y_j \cdot evalh(h_1,\mathscr{N}^{(k)}). \end{aligned}$$

We may assume, therefore, that $j = k$. If i is even, then

$$\begin{aligned} & y_k^i evalh(p_0, \mathscr{N}^{(k)}) + evalh(q_0, \mathscr{N}^{(k+1)}) \\ = \;& (y_k^2)^{\frac{i}{2}} evalh(p_0, \mathscr{N}^{(k)}) + evalh(q_0, \mathscr{N}^{(k+1)}) \\ \equiv \;& evalh(\Theta, \mathscr{N}^{(k)})^{\frac{i}{2}} evalh(p_0, \mathscr{N}^{(k)}) + evalh(\texttt{(POP 1 } q_0\texttt{)}, \mathscr{N}^{(k)}) \\ = \;& evalh(h_0, \mathscr{N}^{(k)}), \end{aligned}$$

and similarly,

$$y_j \left(y_k^i evalh(p_1, \mathscr{N}^{(k)}) + evalh(q_1, \mathscr{N}^{(k+1)}) \right) = y_j \cdot evalh(h_1, \mathscr{N}^{(k)}).$$

if i is odd, then we may rearrange the above expression for $evalh(h, \mathscr{N}^{(k)})$ as

$$\begin{aligned} & \left((y_j^2)^{\frac{i+1}{2}} evalh(p_1, \mathscr{N}^{(k)}) + evalh(q_0, \mathscr{N}^{(k+1)}) \right) \\ + \;\; & y_j \left((y_j^2)^{\frac{i-1}{2}} evalh(p_0, \mathscr{N}^{(k)}) + evalh(q_1, \mathscr{N}^{(k+1)}) \right), \end{aligned}$$

which similarly reduces to $evalh(h_0, \mathscr{N}^{(k)}) + y_j \cdot evalh(h_1, \mathscr{N}^{(k)})$. □

Definition 5.4 *If $h \in \mathscr{H}$, $j \in \{0,1,2\}$, and $(h_0, h_1) = split(h, j, 0)$, then*

$$rewrite(h, j) = h_0 \oplus (h_1 \otimes norm(\mathtt{Y}j, \mathscr{V})).$$

Lemma 5.3 *If $h \in \mathscr{H}$, $j \in \{0,1,2\}$, and $r = rewrite(h, j)$, then $\hat{r} \equiv \hat{h} \pmod{\wp}$.*

PROOF: We instantiate Lemma 5.2 with $k = 0$ and invoke Lemma 4.2. □

Definition 5.5 *If $\sigma \in \mathscr{T}$, then*

$$reduce(\sigma) = rewrite(rewrite(rewrite(norm(\sigma, \mathscr{V}), 0), 1), 2).$$

Lemma 5.4 *If $reduce(\sigma) = reduce(\tau)$, then $\hat{\sigma} \equiv \hat{\tau} \pmod{\wp}$.*

PROOF: This is a consequence of Lemmas 5.3 and 4.3). □

6 Encoding Points on the Curve as Term Triples

The evaluation of terms induces a mapping from $\mathscr{T}^3$ to $\mathbb{F}_\wp^2$:

Definition 6.1 *For $\Pi = (\mu, \nu, \zeta) \in \mathscr{T}^3$, $\hat{\Pi} = (\hat{\mu}, \hat{\nu}, \hat{\zeta})$ and if $\hat{\zeta}$ is not divisible by $\wp$, then $decode(\Pi) = decode(\hat{\Pi})$.*

Clearly, under the following definitions, $decode(\Omega) = O$ and $decode(\Pi_i) = P_i$.

Definition 6.2 *$\Omega = (0,0,1)$ and for $i \in \{0,1,2\}$, $\Pi_i = (\mathtt{X}i, \mathtt{Y}i, 1)$.*

Definition 3.2 suggests a partial addition on $\mathscr{T}^3$ corresponding to the group operation on EC. This in combination with normalization (Definition 4.8) and reduction (Definition 5.5) will provide a practical means of establishing equivalence between expressions constructed from the above points by nested applications of $\oplus$, while avoiding the intractable task of explicitly computing those expressions.

Definition 6.3 *For $\Pi \in \mathscr{T}^3$ and $\Lambda \in \mathscr{T}^3$, $\Pi \oplus \Lambda \in \mathscr{T}^3$ is defined in the following cases:*

(1) *If $\Pi = \Lambda = (\mu, \nu, \zeta)$, then $\Pi \oplus \Lambda = (\mu', \nu', \zeta')$, where*

$$\zeta' = \zeta_{dbl}(\Pi) = \texttt{(* 2 (* } \nu \texttt{ } \zeta \texttt{))},$$

$$\omega = \omega_{dbl}(\Pi) = \texttt{(+ (* 3 (EXPT } \mu \texttt{ 2))}\ \texttt{(+ (* 2 (* } A \texttt{ (* } \mu \texttt{ (EXPT } \zeta \texttt{ 2))))}\ \texttt{(EXPT } \zeta \texttt{ 4))),}$$

$$\mu' = \mu_{dbl}(\Pi) = \texttt{(- (EXPT } \omega' \texttt{ 2)}\ \texttt{(* 4 (* (EXPT } \nu \texttt{ 2) (+ (* } A \texttt{ (EXPT } \zeta \texttt{ 2)) (* 2 } \mu \texttt{))))),}$$

$$\nu' = \nu_{dbl}(\Pi) = \texttt{(- (* } \omega' \texttt{ (- (* 4 (* (EXPT } \nu \texttt{ 2))) } \mu' \texttt{))}\ \texttt{(* 8 (EXPT } \nu \texttt{ 4))).}$$

(2) *If $\Pi = (\theta, \phi, 1)$ and $\Lambda = (\mu, \nu, \zeta) \neq \Pi$, then $\Pi \oplus \Lambda = (\mu', \nu', \zeta')$, where*

$$\zeta' = \zeta_{sum}(\Pi, \Lambda) = \texttt{(* } \zeta \texttt{ (- (* (EXPT } \zeta \texttt{ 2) } \theta \texttt{) } \mu \texttt{),}$$

$$\mu' = \mu_{sum}(\Pi, \Lambda) = \texttt{(- (EXPT (- (* (EXPT } \zeta \texttt{ 3) } \nu \texttt{) 2)}\ \texttt{(* (+ (* (EXPT } \zeta \texttt{ 2) (+ } A \texttt{ } \theta \texttt{)) } \mu \texttt{)}\ \texttt{(EXPT (- (* (EXPT } \zeta \texttt{ 2) } \theta \texttt{) } \mu \texttt{) 2))),}$$

$$\nu' = \nu_{sum}(\Pi, \Lambda) = \texttt{(- (* (- (* (EXPT } \zeta \texttt{ 3) } \phi \texttt{) } \nu \texttt{)}\ \texttt{(- (* (EXPT } \zeta' \texttt{ 2) } \theta \texttt{) } \mu' \texttt{))}\ \texttt{(* (EXPT } \zeta \texttt{ 3) } \phi \texttt{)).}$$

Lemma 6.1 *Let $\Pi \in \mathscr{T}^3$ and $\Lambda \in \mathscr{T}^3$ with decode$(\Pi) = P \in EC$, decode$(\Lambda) = Q \in EC$, and $\Pi \oplus \Lambda$ defined. Assume that if $\Pi = \Lambda$, then $P = Q \neq O$, and otherwise $P \neq Q$. Then decode$(\Pi \oplus \Lambda) = P \oplus Q$.*

PROOF: Let $\Gamma = \Pi \oplus \Lambda$. Clearly, the hypothesis implies that $\hat{\Pi} \oplus \hat{\Lambda}$ is defined. In light of Lemma 3.1, we need only show that $\hat{\Gamma} = \hat{\Pi} \oplus \hat{\Lambda}$.

We shall examine the case $\Pi = \Lambda$; the remaining case is similar. Let $\Pi = (\mu, \nu, \zeta)$, $\Lambda = (\mu', \nu', \zeta')$, and

$$\mathscr{P} = \hat{\Pi} \oplus \hat{\Lambda} = (\hat{\mu}, \hat{\nu}, \hat{\zeta}) = (m, n, z).$$

According to Definition 6.3,

$$\zeta' = \texttt{ (* 2 (* } \nu \texttt{ } \zeta \texttt{))}$$

and it is clear from the definition of *evalp* that

$$\hat{\zeta}' = evalp(\zeta', \mathscr{A}) = 2 \cdot evalp(\nu, \mathscr{A}) \cdot evalp(\zeta, \mathscr{A}) = 2\hat{\nu}\hat{\zeta} = 2mn = z_{dbl}(\mathscr{P}).$$

It may similarly be shown that $\hat{\mu} = m_{dbl}(\mathscr{P})$ and $\hat{\nu} = n_{dbl}(\mathscr{P})$. Thus,

$$\hat{\Gamma} = (\hat{\mu}', \hat{\nu}', \hat{\zeta}') = (z_{dbl}(\mathscr{P}), m_{dbl}(\mathscr{P}), n_{dbl}(\mathscr{P})) = \mathscr{P} \oplus \mathscr{P}. \ \square$$

We also define a negation operator, with the obvious property:

Definition 6.4 *For $\Pi = (\mu, \nu, \zeta) \in \mathscr{T}^3$, $\ominus\Pi = (\mu, \texttt{(- } \nu \texttt{)}, \zeta)$.*

Lemma 6.2 *If* $\Pi \in \mathscr{T}^3$ *and* $decode(\Pi) \in EC$, *then* $decode(\ominus\Pi) = \ominus P$.

The next two lemmas, which combine the results of this section with those of Section 5, will be critical in establishing the group axioms: Lemma 6.3 for closure and Lemma 6.4 for commutativity and associativity.

Definition 6.5 *Given* $\Pi = (\mu, \nu, \zeta) \in \mathscr{T}^3$, *let*

```
τ = (- (EXPT ν 2)
       (+ (EXPT μ 3)
          (+ (* A (EXPT (* μ ζ) 2))
             (* μ (EXPT ζ 4))))).
```

Then Π *is an EC-encoding if* $reduce(\tau) = 0$.

Lemma 6.3 *If* Π *is an EC-encoding and* $P = decode(\Pi)$, *then* $P \in EC$.

PROOF: Let $\Pi = (\mu, \nu, \zeta)$, $\hat{\Pi} = (m,n,z)$, and $P = (x,y) = decode(\Pi)$. Then

$$\hat{\tau} = n^2 - (m^3 + A(mz)^2 + mz^4)$$

and

$$P = \left(\frac{\bar{m}}{\bar{z}^2}, \frac{\bar{n}}{\bar{z}^3}\right).$$

By Lemma 5.4, $\hat{\tau} \equiv 0 \pmod{\wp}$, and therefore, in the field $\mathbb{F}_\wp$,

$$\bar{n}^2 = \bar{m}^3 + A(\bar{m}\bar{z})^2 + \bar{m}\bar{z}^4.$$

Dividing this equation by $\bar{z}^6$ yields

$$y^2 = \frac{\bar{n}^2}{\bar{z}^6} = \frac{\bar{m}^3}{\bar{z}^6} + \frac{A\bar{m}^2}{\bar{z}^4} + \frac{\bar{m}}{\bar{z}^2} = x^3 + Ax^2 + x. \ \square$$

Definition 6.6 *Given* $\Pi = (\mu, \nu, \zeta) \in \mathscr{T}^3$ *and* $\Pi' = (\mu', \nu', \zeta') \in \mathscr{T}^3$, *let*

```
σ = (* μ (EXPT ζ' 2)),    σ' = (* μ' (EXPT ζ 2)),
τ = (* ν (EXPT ζ' 3)),    τ' = (* ν (EXPT ζ 3)).
```

Then $\Pi \sim \Pi' \Leftrightarrow reduce(\sigma) = reduce(\sigma')$ *and* $reduce(\tau) = reduce(\tau')$.

Lemma 6.4 *Let* $\Pi \in \mathscr{T}^3$ *and* $\Pi' \in \mathscr{T}^3$. *If* $decode(\Pi) = P \in EC$, $decode(\Pi') = P' \in EC$, *and* $\Pi \sim \Pi'$, *then* $P = P'$.

PROOF: Let $\Pi = (\mu, \nu, \zeta)$, $\Pi' = (\mu', \nu', \zeta')$, $\hat{\Pi} = (m,n,z)$, and $\hat{\Pi}' = (m',n',z')$. Then by Lemma 5.4,

$$mz'^2 = \hat{\mu}\hat{\zeta}'^2 = \hat{\sigma} \equiv \hat{\sigma}' = \hat{\mu}'\hat{\zeta}^2 = m'z^2 \pmod{\wp}$$

and

$$nz'^3 = \hat{\nu}\hat{\zeta}'^3 = \hat{\tau} \equiv \hat{\tau}' = \hat{\nu}'\hat{\zeta}^3 = n'z^3 \pmod{\wp}.$$

Thus, in the field $\mathbb{F}_\wp$,

$$P = \left(\frac{\bar{m}}{\bar{z}^2}, \frac{\bar{n}}{\bar{z}^3}\right) = \left(\frac{\bar{m}'}{\bar{z}'^2}, \frac{\bar{n}'}{\bar{z}'^3}\right) = P'. \ \square$$

7 Abelian Group Axioms

It must be shown that if $\{P,Q,R\} \subset EC$, then $P \oplus Q = Q \oplus P \in EC$ and $(P \oplus Q) \oplus R = P \oplus (Q \oplus R)$. We may assume that the points are finite, since each of these properties is trivial otherwise, and without loss of generality, we may confine our attention to the fixed points P_0, P_1, and P_2.

Computations 1–11 below are computational results of evaluating the functions that are specified by Definitions 5.5 (term reduction), 6.3 (addition of term triples), 6.4 (negation of a term triple), 6.5 (EC-encoding recognizer), and 6.6 (equivalence of term triples). The lemmas of this section are derived from these results using the corresponding Lemmas 5.4, 6.1, 6.2, 6.3, and 6.4.

Computation 1 $\Pi_0 \oplus \Pi_0$ *and* $\Pi_0 \oplus \Pi_1$ *are EC-encodings.*

Lemma 7.1 (Closure) $P_0 \oplus P_1 \in EC$.

PROOF: If $P_0 \neq P_1$, then by Computation 1, Definition 6.2 and Lemmas 6.1,

$$P_0 \oplus P_1 = decode(\Pi_0) \oplus decode(\Pi_1) = decode(\Pi_0 \oplus \Pi_1),$$

and by Lemma 6.3, $P_0 \oplus P_1 \in EC$. Similarly, $P_0 \oplus P_0 \in EC$. □

Computation 2 $\Pi_0 \oplus \Pi_1 \sim \Pi_1 \oplus \Pi_0$.

Lemma 7.2 (Commutativity) $P_0 \oplus P_1 = P_1 \oplus P_0$.

PROOF: We may assume $P_0 \neq P_1$. By Computation 2 and Lemmas 6.1 and 6.4,

$$P_0 \oplus P_1 = decode(\Pi_0 \oplus \Pi_1) = decode(\Pi_1 \oplus \Pi_0) = P_1 \oplus P_0.$$ □

All remaining results pertain to associativity.

Computation 3 $\ominus(\Pi_0 \oplus \Pi_0) \sim (\ominus\Pi_0) \oplus (\ominus\Pi_0)$.

Computation 4 $\ominus(\Pi_0 \oplus \Pi_1) \sim (\ominus\Pi_0) \oplus (\ominus\Pi_1)$.

Lemma 7.3 $\ominus(P_0 \oplus P_1) = (\ominus P_0) \oplus (\ominus P_1)$.

PROOF: This follows from Computations 3 and 4 and Lemmas 5.4, 6.1, 6.2, and 6.4. □

Computation 5 $(\ominus\Pi_0) \oplus (\Pi_0 \oplus \Pi_0) \sim \Pi_0$.

Computation 6 $(\ominus\Pi_0) \oplus (\Pi_0 \oplus \Pi_1) \sim \Pi_1$.

Lemma 7.4 *If* $P_0 \oplus P_1 \neq \ominus P_0$, *then* $(\ominus P_0) \oplus (P_0 \oplus P_1) = P_1$.

PROOF: This follows similarly from Computations 5 and 6. □

Computation 7 $\Pi_2 \oplus (\Pi_0 \oplus \Pi_1) \sim \Pi_1 \oplus (\Pi_0 \oplus \Pi_2)$.

Computation 8 $\Pi_1 \oplus (\Pi_0 \oplus \Pi_0) \sim \Pi_0 \oplus (\Pi_0 \oplus \Pi_1)$.

Lemma 7.5 *If* $P_0 \oplus P_1 \notin \{P_2, \ominus P_2\}$ *and* $P_0 \oplus P_2 \notin \{P_1, \ominus P_1\}$, *then*

$$P_2 \oplus (P_0 \oplus P_1) = P_1 \oplus (P_0 \oplus P_2).$$

PROOF: The claim follows immediately from Computations 7 and 8 and Lemmas 5.4, 6.1, and 6.4 except in the cases $P_0 = \ominus P_1$, $P_0 = P_1 = O$, $P_0 = \ominus P_2$, and $P_0 = P_2 = O$. We need only consider the first two of these cases; the other two are similar. Moreover, since $\ominus O = O$, the second case is subsumed by the first. Thus, we may assume $P_0 = \ominus P_1$. Now *LHS* (the left-hand side) is P_2 and by Lemma 7.4, $RHS = (\ominus P_0) \oplus (P_0 \oplus P_2) = P_2 = LHS$. □

Computation 9 $(\Pi_0 \oplus \Pi_0) \oplus (\Pi_0 \oplus \Pi_0) \sim \Pi_0 \oplus (\Pi_0 \oplus (\Pi_0 \oplus \Pi_0))$.

Computation 10 $(\Pi_0 \oplus \Pi_1) \oplus (\Pi_0 \oplus \Pi_1) \sim \Pi_0 \oplus (\Pi_1 \oplus (\Pi_0 \oplus \Pi_1))$.

Lemma 7.6 *If* $P_0 \oplus P_1 \neq -(P_0 \oplus P_1)$, $P_0 \oplus P_1 \neq \ominus P_1$, *and* $P_1 \oplus (P_0 \oplus P_1) \notin \{P_0, \ominus P_0\}$, *then* $(P_0 \oplus P_1) \oplus (P_0 \oplus P_1) = P_0 \oplus (P_1 \oplus (P_0 \oplus P_1))$.

PROOF: The case $P_0 = \ominus P_1$ is trivial and the case $P_1 = P_0 \oplus P_1$ is precluded by Lemma 2.1. All other cases are handled by Computations 9 and 10 and Lemmas 5.4, 6.1, and 6.4. □

Computation 11 *Let* $\Sigma = \Pi_0 \oplus \Pi_1 = (\mu, \nu, \zeta)$, $\Sigma' = \Pi_0 \oplus \Pi_0 = (\mu', \nu', \zeta')$,

$\phi =$
```
(- (EXPT (+ (- μ (* X1 (EXPT ζ 2))) (* 2 (* Y1 Y2))) 2)
   (EXPT (* 2 (* Y1 Y2)) 2)),
```

and

$\psi =$
```
(* (- μ' (* X2 (EXPT ζ' 2))) (EXPT ζ 2)).
```

Then $reduce(\phi) = reduce(\psi)$.

Lemma 7.7 *If* $P_0 \oplus P_1 = \ominus P_0$, *then* $P_1 = \ominus(P_0 \oplus P_0)$.

PROOF: First note that we may assume that $P_0 \notin \{P_1, \ominus P_1\}$, for if $P_0 = P_1$, then

$$P_1 = \ominus(\ominus P_0) = \ominus(P_0 \oplus P_1) = \ominus(P_0 \oplus P_0),$$

and if $P_0 = \ominus P_1$, then

$$\ominus P_0 = P_0 \oplus P_1 = (\ominus P_1) \oplus P_1 = \infty,$$

contradicting $P_0 \neq \infty$. Furthermore, if $P_0 = \ominus P_0$, then $P_0 \oplus P_1 = P_0$, contradicting Lemma 2.1. Thus, we have $x_0 \neq x_1$ and $y_0 \neq 0$.

Retaining the notation of Computation 11, let

$$\hat{\Sigma} = (\hat{\mu}, \hat{\nu}, \hat{\zeta}) = (m, n, z)$$

and

$$\hat{\Sigma}' = (\hat{\mu}', \hat{\nu}', \hat{\zeta}') = (m', n', z').$$

It follows from the definition of *evalp* that

$$\hat{\phi} = (m - x_0 z^2 + 2y_0 y_1)^2 - (2y_0 y_0)^2$$

and

$$\hat{\psi} = (m' - x_1 z'^2) z^2.$$

By Lemma 6.1,

$$decode(\Sigma) = \left(\frac{\bar{m}}{\bar{z}^2}, \frac{\bar{n}}{\bar{z}^3}\right) = P_0 \oplus P_1 = \ominus P_0 = (x_0, -y_0),$$

and hence $m \equiv x_0 z^2 \pmod{\wp}$, which implies $\hat{\phi} \equiv 0 \pmod{\wp}$.

By Computation 11 and Lemma 5.4, $\hat{\psi} \equiv 0 \pmod{\wp}$, which implies $m' \equiv x_1 z'^2 \pmod{\wp}$. Thus, by Lemma 6.1,

$$P_0 \oplus P_0 = decode(\Sigma') = \left(\frac{\bar{m'}}{\bar{z'}^2}, \frac{\bar{n'}}{\bar{z'}^3}\right) = \left(x_1, \frac{\bar{n'}}{\bar{z'}^3}\right),$$

which implies $P_0 \oplus P_0 \in \{P_1, \ominus P_1\}$. We need only consider the case $P_0 \oplus P_0 = P_1$.

Suppose that $P_0 \oplus P_0 = P_1$. Then $P_0 \oplus P_0 \neq \ominus P_0$, and by Lemma 7.10,

$$P_1 \oplus (\ominus P_0) = (P_0 \oplus P_0) \oplus (\ominus P_0) = P_0.$$

Thus,

$$P_0 \oplus (\ominus P_1) = \ominus(\ominus P_0 \oplus P_1) = \ominus(P_1 \oplus (\ominus P_0)) = \ominus P_0 = P_0 \oplus P_1.$$

By Lemma 2.2, $P_1 = O$, and hence $P_1 = \ominus P_1 = \ominus(P_0 \oplus P_0)$. □

Lemma 7.8 *If $P_0 \oplus P_1 = \ominus P_2$, then $(P_0 \oplus P_1) \oplus P_2 = P_0 \oplus (P_1 \oplus P_2)$.*

PROOF: $LHS = \infty$ and by Lemma 7.3,

$$RHS = P_0 \oplus (P_1 \oplus (\ominus(P_0 \oplus P_1))) = P_0 \oplus (P_1 \oplus ((\ominus P_0) \oplus (\ominus P_1))).$$

Therefore, we must show that $P_1 \oplus ((\ominus P_0) \oplus (\ominus P_1)) = \ominus P_0$.

If $(\ominus P_0) \oplus (\ominus P_1) \neq P_1$, then this follows from Lemma 7.4. On the other hand, if $(\ominus P_0) \oplus (\ominus P_1) = P_1$, then by Lemmas 7.7 and 7.3,

$$\ominus P_0 = \ominus((\ominus P_1) \oplus (\ominus P_1)) = P_1 \oplus P_1 = P_1 \oplus ((\ominus P_0) \oplus (\ominus P_1)).$$ □

Lemma 7.9 $(P_0 \oplus P_0) \oplus P_1 = P_0 \oplus (P_0 \oplus P_1)$.

PROOF: By Lemma 7.8, we may assume $P_0 \oplus P_1 \neq \ominus P_0$ and $P_0 \oplus P_0 \neq \ominus P_1$. By Lemmas 2.1 and 7.5, we may assume that $P_1 = P_0 \oplus P_1$.

If $P_1 = \ominus P_0$, then

$$LHS = P_1 + P_1 = (\ominus P_0) \oplus (\ominus P_0) = \ominus(P_0 \oplus P_0) = \ominus P_1 = P_0 = RHS.$$

But if $P_1 \neq \ominus P_0$, then the claim follows from Lemma 7.6. □

Two final computations are required for the case $P_1 = O$ and $P_2 = P_0 \oplus P_1$:

Computation 12 $(\Pi_0 \oplus \Omega) \oplus (\Pi_0 \oplus \Omega) \sim \Pi_0 \oplus \Pi_0$.

Computation 13 $\Omega \oplus (\Pi_0 \oplus \Omega) \sim \Pi_0$.

Lemma 7.10 $(P_0 \oplus O) \oplus (P_0 \oplus O) = P_0 \oplus (O \oplus (P_0 \oplus O))$.

PROOF: We may assume that $P_0 \neq O$. Since Lemma 2.1 implies $P_0 \oplus O \neq O$, it follows from Computation 13 that $O \oplus (P_0 \oplus O) = P_0$. Thus, the claim reduces to $(P_0 \oplus O) \oplus (P_0 \oplus O) = P_0 \oplus P_0$, which follows from Computation 12. □

Lemma 7.11 (Associativity) $(P_0 \oplus P_1) \oplus P_2 = P_0 \oplus (P_1 \oplus P_2)$.

PROOF: By Lemmas 7.6 and 7.8, we may assume $P_1 \oplus P_2 \neq \ominus P_0$, and $P_2 = P_0 \oplus P_1$. By Lemma 7.6, we need only eliminate the cases $P_0 \oplus P_1 = \ominus P_1$ and $P_1 \oplus (P_0 \oplus P_1) = P_0$.

If $P_2 = P_0 \oplus P_1 = \ominus P_1$, then $RHS = P_0$ and by Lemmas 7.3 and 7.7,

$$LHS = (P_0 \oplus P_1) \oplus (P_0 \oplus P_1) = (\ominus P_1) \oplus (\ominus P_1) = \ominus(P_1 \oplus P_1) = P_0 = RHS.$$

Finally, if $P_1 \oplus (P_0 \oplus P_1) = P_0$, then Lemma 7.9 implies $P_0 = P_0 \oplus (P_1 \oplus P_1)$, Lemma 2.1 then implies $P_1 = O$, and the claim follows from Lemma 7.10. □

References

[1] Henk Barendregt & Erik Barendsen (2002): *Autarchic Computations in Formal Proofs*. Journal of Automated Reasoning 28, pp. 321–336, doi:10.1023/A:1015761529444.

[2] Daniel J. Bernstein (2006): *Curve25519: New Diffie-Hellman Speed Records*. In: 9th International Conference on Theory and Practice of Public Key Cryptography, Springer, pp. 207–228, doi:10.1007/11745853_14.

[3] Daniel J. Bernstein & Tanja Lange (2011): *A Complete Set of Addition Laws for Incomplete Edwards Curves*. Journal of Number Theory 131, pp. 858–872, doi:10.1016/j.jnt.2010.06.015.

[4] Stefan Friedl (1998): *An Elementary Proof of the Group Law for Elliptic Curve*.

[5] Benjamin Gregoire & Assia Mahboubi (2005): *Proving Equalities in a Commutative Ring Done Right in Coq*. In: Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics, Springer-Verlag, pp. 98–113, doi:10.1007/11541868_7.

[6] Henri Poincaré (1901): *Sur les Propriétés Arithmetiques des Courbes Algébriques*. Lournalde Mathématiques Pures et Appliuées 7, pp. 121–233.

[7] Vaughn Pratt (1975): *Every Prime Has a Succinct Certification*. SIAM Journal on Computing 4, doi:10.1137/0204018.

[8] David M. Russinoff: *Polynomial Terms and Sparse Horner Normal Form*. Available at `http://www.russinoff.com/papers/shnf.pdf`.

[9] David M. Russinoff: *Pratt Certification and the Primality of* $2^{255} - 19$. Available at `http://www.russinoff.com/papers/pratt.pdf`.

[10] Joseph H. Silverman & John T. Tate (2015): *Rational Points on Elliptic Curves*. Springer-Verlag, doi:10.1007/978-3-319-18588-0.

[11] Laurent Théry (2007): *Proving the Group Law for Elliptic Curves Formally*. Technical Report RT-0330, Inria.

[12] Thomas Tymoczko (1998): *Computers and Mathematical Practice: A Case Study*. Princeton University Press.

Meta-extract: Using Existing Facts in Meta-reasoning

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

Sol Swords
Centaur Techology, Inc.
Austin, TX, USA
sswords@centtech.com

ACL2 has long supported user-defined simplifiers, so-called *metafunctions* and *clause processors*, which are installed when corresponding rules of class `:meta` or `:clause-processor` are proved. Historically, such simplifiers could access the logical world at execution time and could call certain built-in proof tools, but one could not assume the soundness of the proof tools or the truth of any facts extracted from the world or context when proving a simplifier correct. Starting with ACL2 Version 6.0, released in December 2012, an additional capability was added which allows the correctness proofs of simplifiers to assume the correctness of some such proof tools and extracted facts. In this paper we explain this capability and give examples that demonstrate its utility.

1 Introduction

The meta rule and clause processor facilities of the ACL2 theorem proving system [6] are designed to allow users to write custom proof routines which, once proven correct, can be called by the ACL2 prover[1]. These facilities descend from the meta reasoning capability provided by earlier Boyer-Moore provers [3]. Relevant background can be found in earlier ACL2 papers on meta reasoning [5] and clause processors [7], along with ACL2 documentation [1].

Such meta-level proof routines have historically been limited in their use of ACL2's database of stored facts and its built-in prover functions. For example, a metafunction could call the ACL2 rewriter, but when proving this metafunction correct, the result of calling the rewriter could not be assumed to be equivalent to the input — that is, the rewriter could not be assumed to be correct. Similarly, metafunctions and clause processors could both examine the ACL2 world (the logical state of the prover, including the database of stored facts), but in proving the correctness of these functions, facts extracted from the world could not be assumed to be correct.

Meta-extract is an ACL2 feature first introduced in Version 6.0 (December, 2012). It allows for certain facts stored in the ACL2 world and certain ACL2 prover routines to be assumed correct when proving the correctness of metafunctions and clause processors. In particular, additional *meta-extract hypotheses* that capture the correctness of such routines and facts are allowed to be present in these correctness theorems. These additional hypotheses preserve the soundness of metareasoning in ACL2 because they encode assumptions that we are already making — that the facts stored in the ACL2 logical world are true and that ACL2's prover routines are sound.

Note that meta-extract hypotheses can be used to help *prove* theorems to be stored as meta rules or as clause-processor rules, but they have no effect on how those rules are *applied* during subsequent proofs. For that, we refer readers to existing papers [5, 7] as well as documentation [1] topics including META, CLAUSE-PROCESSOR, and EXTENDED-METAFUNCTIONS. Also see the topic META-EXTRACT

[1] Clause processors may alternatively be "trusted" — used without proof — but at the cost of a *trust tag* marking this use as a source of potential unsoundness. [7]

Warren Hunt Jr. and Anna Slobodova (Eds.):
ACL2 Theorem Prover and its Applications
EPTCS 249, 2017, pp. 47–60, doi:10.4204/EPTCS.249.4

for additional user-level documentation. Note that references to documentation topics, such as those above, are underlined to denote hyperlinks to topics in the online documentation [1] for ACL2 and its books.

This paper primarily addresses metafunctions rather than always referring to both metafunctions and clause processors. The correctness arguments are analogous. The meta-extract functionality available for clause processors is a subset of the functionality available for metafunctions: the *global facts* discussed below are available for both, but *contextual facts* need contexts that are only available below the clause level.

1.1 Meta-extract Hypotheses

We next introduce the various meta-extract hypotheses and explain informally why it is sound to include them. See the *Essay on Correctness of Meta Reasoning* in the ACL2 source code for a rigorous mathematical argument.

Correctness theorems for metafunctions are stated using a function that we will call a *pseudo-evaluator*, typically defined via the `defevaluator` macro. (Elsewhere in the ACL2 literature this is simply referred to as an evaluator; however, we want to emphasize the difference between a pseudo-evaluator and a real term evaluator.) A pseudo-evaluator is a constrained function about which only certain facts are known; these facts are ones that would also be true of a "real" evaluator capable of fully interpreting any ACL2 term. The intention is for these facts to suffice for proving a metafunction correct, without the need for a real term evaluator[2]. To prove a metafunction correct the user must show that the pseudo-evaluation of the term output by the metafunction is equal (or equivalent) to the pseudo-evaluation of the input term. Intuitively, if this can be proved of a pseudo-evaluator, and the only facts known about the pseudo-evaluator are ones that are also true of the real evaluator of ACL2 terms, then this must also be true of the real evaluator: that is, the evaluations of the input and output terms of the metafunction are equivalent, and thus the metafunction is correct.

The meta-extract feature allows certain additional hypotheses in the statement of the correctness theorem of a metafunction. These meta-extract hypotheses are applications of the pseudo-evaluator to calls of either the function `meta-extract-global-fact+`, its less general version `meta-extract-global-fact`, or the function `meta-extract-contextual-fact`. These functions produce various sorts of terms by extracting facts from the ACL2 world and calling ACL2 prover subroutines, constructed so that if ACL2 and its logical state are sound, the terms produced should always be true. For example, these functions can produce:

- the body of a previously proven theorem;
- the definitional equation of a previously defined function;
- a term equating a call of a function on quoted constants to the quoted value of that call;
- a term equating some term *a* to the result of rewriting *a* using `mfc-rw`; or
- a term describing the type of *a*, according to ACL2's type-set reasoning.

When we use such facts in our metafunction, meta-extract hypotheses allow us to assume that they are true according to the pseudo-evaluator while doing the correctness proof.

[2] It would be unsound to define what we are calling a real evaluator in ACL2 — in particular, one that could evaluate terms containing calls of the evaluator itself, or functions that call the evaluator. In order to fix our handwavy argument, you could instead think of introducing an evaluator that can interpret all functions that are defined at the point when the metafunction is being applied.

Intuitively, adding meta-extract hypotheses to a metafunction's correctness theorem is allowable because we expect the (real) evaluation of any term produced by one of the meta-extract functions to return true. If we prove the pseudo-evaluator theorem with meta-extract hypotheses and, as before, reason that since the theorem is true of the pseudo-evaluator, it is also true of the real evaluator, then the final step is to observe that meta-extract hypotheses are true using the real evaluator (or else ACL2 is already unsound). Therefore, even with meta-extract hypotheses, we can still conclude that the evaluations of the metafunction's output and input terms are equivalent.

Why not somehow axiomatize the idea that the contents of the ACL2 world are correct? Simply put, we don't see how to do that, and ideas we have heard along those lines have been unsound. At the least, one would seem to need to formalize the complex notion of a valid world.

1.2 Organization of This Paper

In Section 2 we provide examples of meta-extract hypotheses and summary documentation of their various forms. Section 3 presents a community book that provides a convenient way to use the meta-extract facility. Next we present applications in Section 4 and finally, we conclude with Section 5.

2 Meta-extract

Next we explain meta-extract by first giving two examples and then summarizing the general forms of meta-extract hypotheses.

2.1 Tutorial Examples

We present two examples for the two kinds of meta-extract hypotheses, corresponding to evaluation of calls of `meta-extract-contextual-fact` and of `meta-extract-global-fact`. (Since a call `(meta-extract-global-fact obj state)` is an abbreviation for `(meta-extract-global-fact+ obj state state)`, we are thus effectively illustrating `meta-extract-global-fact+` as well.)

2.1.1 Meta-extract-contextual-fact

Our first example is intentionally contrived and quite trivial, intended only to provide an easy introduction to meta-extract. It illustrates the use of `meta-extract-contextual-fact`. The intent is to simplify any term of the form `(nth` *x lst*`)`, when *x* is easily seen by ACL2 to be a symbol in the current context, to `(car` *lst*`)`.

In ACL2, the use of metafunctions is always supported by an evaluator, called a *pseudo-evaluator* in the preceding section. Let us introduce an evaluator that "knows" about the functions relevant to this example.

```
(defevaluator nthmeta-ev nthmeta-ev-lst
  ((typespec-check ts x)
   (nth n x)
   (car x)))
```

Next we define a metafunction, intended to replace any term `(nth n x)` by a corresponding term `(car x)` when `n` is known to be a symbol using type-set reasoning.[3]

[3] For details such as the meaning of `:forcep nil`, see the documentation for meta-extract.

```
(defun nth-symbolp-metafn (term mfc state)
  (declare (xargs :stobjs state))
  (case-match term
    (('nth n x)
     (if (equal (mfc-ts n mfc state :forcep nil)
                *ts-symbol*)
         (list 'car x)
       term))
    (& term)))
```

When the input term matches `(nth` *n x*`)`, this calls `mfc-ts` to deduce the possible types of *n*. If that type-set equals `*ts-symbol*` then that term must evaluate to a symbol, and the term can be reduced to `(car` *x*`)`.[4]

Now we can present a meta rule with a meta-extract hypothesis. Without that hypothesis the formula below is not a theorem, because the function `mfc-ts` has no axiomatic properties; all we know about it below is what we are told by the meta-extract hypothesis, as discussed further below.

```
(defthm nth-symbolp-meta
    (implies (nthmeta-ev (meta-extract-contextual-fact `(:typeset ,(cadr term))
                                                       mfc
                                                       state)
                         a)
             (equal (nthmeta-ev term a)
                    (nthmeta-ev (nth-symbolp-metafn term mfc state) a)))
    :rule-classes ((:meta :trigger-fns (nth))))
```

To see what the meta-extract hypothesis above gives us, consider the following theorem provable by ACL2.

```
(equal (meta-extract-contextual-fact `(:typeset ,x)
                                     mfc
                                     state)
       (list 'typespec-check
             (list 'quote
                   (mfc-ts x mfc state :forcep nil))
             x))
```

At a high level, this theorem shows us that `meta-extract-contextual-fact` returns a term of the form `(typespec-check (quote` *ts*`) x)`, which asserts that the term `x` belongs to the set of values represented by *ts*. The meta-extract hypothesis applies the pseudo-evaluator to this term, and since `typespec-check` is one of its known functions, the hypothesis reduces to

```
(typespec-check (mfc-ts (cadr term) mfc state :forcep nil)
                (nthmeta-ev (cadr term) a)).
```

The interesting case in proving our metafunction correct is when this `mfc-ts` call equals `*ts-symbol*`. In this case the hypothesis becomes

[4] Requiring the type-set to equal `*ts-symbol*` is an an unnecessarily strong check, chosen merely for ease of presentation: it requires that ACL2's determination of the term's possible types include all three of the basic types `T`, `NIL`, and non-Boolean symbols.

```
(typespec-check *ts-symbol* (nthmeta-ev (cadr term) a))
```

which, when expanded, implies that the evaluation of `(cadr term)` must be a symbol, which enables the proof of `nth-symbolp-meta`.

Recall that meta-extract hypotheses do not affect the *applications* of meta rules; they only support their proofs. Therefore, the following test of the example above yields no surprises.

```
(defstub foo (x) t)
(thm (implies (symbolp (foo x))
              (equal (nth (foo x) y) (car y)))
     :hints (("Goal" :in-theory '(nth-symbolp-meta))))
```

2.1.2 Meta-extract-global-fact

Our second example is from community book "demos/nth-update-nth-meta-extract.lisp", which uses `meta-extract-global-fact`. Let us begin by considering what problem this book is attempting to solve.

Consider a `defstobj` event, `(defstobj st` $\texttt{fld}_1$ $\texttt{fld}_2$ `...` $\texttt{fld}_n$`)`. A common challenge in reasoning about stobjs is the simplification of *read-over-write* terms, of the form `(`$\texttt{fld}_i$ `(update-`$\texttt{fld}_j$ v `st))`, which indicate that we are to read $\texttt{fld}_i$ after updating $\texttt{fld}_j$. That term simplifies to `(`$\texttt{fld}_i$ `st)` when $i \neq j$, and otherwise it simplifies to v. How do we get ACL2 to do such simplification automatically? The following two approaches are standard.

- Disable the stobj accessors and updaters after proving rewrite rules to simplify terms of the form `(`$\texttt{fld}_i$ `(update-`$\texttt{fld}_j$ `val st))`, to `val` if $i = j$ and to `(`$\texttt{fld}_i$ `st)` if $i \neq j$.
- Let the stobj accessors and updaters remain enabled, relying on a rule such as the built-in rewrite rule `nth-update-nth` to rewrite terms, obtained after expanding calls of the accessors and updaters, of the form `(nth` i `(update-nth` j `val st))`.

The first of these requires n^2 rules, which is generally feasible but can perhaps get somewhat unwieldy. The second of these provides a simple solution, but when proofs fail, the resulting checkpoints can be more difficult to comprehend.

Here, we outline a solution that addresses both of these concerns: a macro that generates a suitable meta rule. Details of this proof development may be found in community book "demos/nth-update-nth-meta-extract.lisp". First we introduce our metafunction. Next, we prove a meta rule for a specific stobj. Finally, we mention a macro that generates a version of this rule that is suitable for an arbitrary specified stobj.

Our metafunction returns the input term unchanged unless it is of the form `(`r `(`w v x`))`, where r and w are *reader* (accessor) and *writer* (updater) functions defined to be calls of `nth` and `update-nth`, on explicit indices: `(`r x`)` = `(nth '`i x`)` and `(`w v x`)` = `(update-nth '`i v x`)`. In that case, the function `nth-update-nth-meta-fn-new-term` computes a new term: v if $i = j$, and otherwise, `(`r x`)`.

```
(defun nth-update-nth-meta-fn (term mfc state)
  (declare (xargs :stobjs state)
           (ignore mfc))
  (or (nth-update-nth-meta-fn-new-term term state)
      term))
```

Notice below that in computing the new term, the definitions of the reader and writer are extracted from the logical world using the function, `meta-extract-formula`, which returns the function's definitional equation. For example:

```
ACL2 !>(meta-extract-formula 'atom state)
(EQUAL (ATOM X) (NOT (CONSP X)))
ACL2 !>
```

We thus rely on the correctness of `meta-extract-formula` for the equality of the input term and the term returned by the following function.

```
(defun nth-update-nth-meta-fn-new-term (term state)
  (declare (xargs :stobjs state))
  (case-match term
    ((reader (writer val x))
     (and (not (eq reader 'quote))
          (not (eq writer 'quote))
          (let* ((reader-formula (and (symbolp reader)
                                      (meta-extract-formula reader state)))
                 (i-rd (fn-nth-index reader reader-formula)))
            (and
             i-rd  ; the body of reader is (nth 'i-rd ...)
             (let* ((writer-formula (and (symbolp writer)
                                         (meta-extract-formula writer state)))
                    (i-wr (fn-update-nth-index writer writer-formula)))
               (and
                i-wr  ; the body of writer is (update-nth 'i-wr ...)
                (if (eql i-rd i-wr)
                    val
                  (list reader x))))))))
    (& nil)))
```

Next we introduce a (pseudo-)evaluator to use in our meta rule.

```
(defevaluator nth-update-nth-ev nth-update-nth-ev-lst
  ((nth n x)
   (update-nth n val x)
   (equal x y)))
```

We need to define one more function before presenting our meta rule. It takes as input a term $(f\ t_1\ \ldots\ t_n)$ with list of formals $(v_1\ \ldots\ v_n)$, and builds an alist that maps each v_i to the value of t_i in a given alist. Like our metafunction, it consults `meta-extract-formula` to obtain the formal parameters of f. An earlier attempt to use the function, `formals`, failed: the meta rule's proof needs these formals to connect to those found by our metafunction.

```
(defun meta-extract-alist-rec (formals actuals a)
  (cond ((endp formals) nil)
        (t (acons (car formals)
                  (nth-update-nth-ev (car actuals) a)
                  (meta-extract-alist-rec (cdr formals) (cdr actuals) a)))))

(defun meta-extract-alist (term a state)
```

```
  (declare (xargs :stobjs state :verify-guards nil))
  (let* ((fn (car term))
         (actuals (cdr term))
         (formula (meta-extract-formula fn state))  ; (equal (fn ...) ...)
         (formals (cdr (cadr formula))))
    (meta-extract-alist-rec formals actuals a)))
```

We now define a stobj and prove a corresponding meta rule.

```
(defstobj st
  fld1 fld2 fld3 fld4 fld5 fld6 fld7 fld8 fld9 fld10
  fld11 fld12 fld13 fld14 fld15 fld16 fld17 fld18 fld19 fld20)

(defthm nth-update-nth-meta-rule-st
  (implies
   (and (nth-update-nth-ev  ; (f (update-g val st))
         (meta-extract-global-fact (list :formula (car term)) state)
         (meta-extract-alist term a state))
        (nth-update-nth-ev  ; (update-g val st)
         (meta-extract-global-fact (list :formula (car (cadr term)))
                                   state)
         (meta-extract-alist (cadr term) a state))
        (nth-update-nth-ev  ; (f st) -- note st is (caddr (cadr term))
         (meta-extract-global-fact (list :formula (car term)) state)
         (meta-extract-alist (list (car term)
                                   (caddr (cadr term)))
                             a state)))
   (equal (nth-update-nth-ev term a)
          (nth-update-nth-ev (nth-update-nth-meta-fn term mfc state) a)))
  :hints ...
  :rule-classes ((:meta :trigger-fns (fld1 fld2 ... fld20))))
```

The proof of the theorem below takes no measurable time, and applies the metafunction proved correct above.

```
(in-theory (disable fld1 ... fld20 update-fld1 ... update-fld20))

(defthm test1
  (equal (fld3 (update-fld1 1
                (update-fld2 2
                 (update-fld3 3
                  (update-fld4 4
                   (update-fld3 5
                    (update-fld6 6 st)))))))
         3))
```

Notice that there is nothing about the meta rule above that is specific to the particular stobj, `st`, except for the `:trigger-fns` that it specifies. In the community book mentioned above ("demos/nth-update-nth-meta-extract.lisp"), we define a macro that automates the generation of such a meta rule for an arbitrary stobj. Our macro takes the name of a stobj, *s*, and does two things: it disables all of the stobj's accessors and updaters, and it proves a meta rule that simplifies every term of the form `(`*r* `(`*w* *v* *s*`))`, where *r* and *w* are an accessor and updater, respectively, for the stobj *s*.

The meta rule above uses three meta-extract-global-fact hypotheses corresponding to uses of three particular facts. Enumerating all of the facts potentially used in the metafunction in this way could easily become unwieldy; in Section 3 we discuss a utility that solves this problem, allowing one meta-extract-global-fact hypothesis and one meta-extract-contextual-fact hypothesis to cover all of the facts that might be used.

2.2 General Forms

In this section we summarize briefly the forms of meta-extract hypotheses. Additional details may be found in the documentation for meta-extract.

Below, let evl be the pseudo-evaluator (see Section 1.1) used in a meta rule. The two primary forms of meta-extract hypotheses that can be used in a meta rule are as follows. In the first, aa represents an arbitrary term; in the second, a must be the second argument of the two calls of the pseudo-evaluator in the conclusion of the theorem.

```
(evl (meta-extract-global-fact obj state) aa)
(evl (meta-extract-contextual-fact obj mfc state) a)
```

The second form above is only legal for meta rules about extended metafunctions (which take arguments mfc and state). The first form above is actually equivalent to the first form below, which in turn is a special case of the second form below.

```
(evl (meta-extract-global-fact+ obj state state) aa)
(evl (meta-extract-global-fact+ obj st state) aa)
```

The last form supports clause processors that modify state as long as they do not change the logical world; it produces the same value as the previous form as long as both st and state have equal world fields.

If the arguments to the meta-extract-* function are somehow malformed, then it returns the trivial term 'T, which is not of any use in proving a metafunction correct. Otherwise, each invocation produces a term that states the "correctness" of an invocation of some utility, such as mfc-rw+ or meta-extract-formula. The terms produced for different kinds of invocation vary according to the particular concept of correctness appropriate to the utility in question.

We now describe the allowed obj arguments to meta-extract-global-fact and the terms they produce. The documentation for extended-metafunctions explains meta-level functions discussed below, such as mfc-rw and mfc-ap.

- (:formula *f*) produces the value of (meta-extract-formula *f* state), which allows metafunctions to assume that any invocation of meta-extract-formula produces a true formula. Meta-extract-formula looks up various kinds of formulas from the world:
  - the body of *f* if it is a theorem name
  - the definitional equation of *f* if it is a defined function
  - the constraint of *f* if it is a constrained function
  - the defchoose axiom of *f* if it is a defchoose function.
- (:lemma *f n*) produces a term corresponding to the *n*th rewrite rule stored in the lemmas property of function *f*, which allows any such rule to be assumed correct in a metafunction. The term returned is the value of

`(rewrite-rule-term (nth 'n (getpropc 'f 'lemmas nil (w state))))`,

assuming that *n* is a valid index (does not exceed the length of the indicated list). Here `rewrite-rule-term` transforms a `rewrite-rule` record structure into a term such as

`(implies hyps (equiv lhs rhs))`.

The rewrite rules stored in `:lemmas` are not simply theorem bodies, which could be accessed using `meta-extract-formula`, but rewrite rule structures, which separately store the left-hand side, right-hand side, equivalence relation, and hypotheses, and also contain heuristic information such as the backchain limits and match-free mode.

- `(:fncall f L)` produces a term `(equal c 'v)`, where *c* is the call that applies *f* to the quotations of the values in argument list *L*, and *v* is the value of that call computed by `magic-ev-fncall`. This allows metafunctions to assume that `magic-ev-fncall` correctly evaluates function applications.

The allowed `obj` arguments to `meta-extract-contextual-fact` are as follows.

- `(:typeset term)` produces a term stating the correctness of the type-set produced by `mfc-ts` for *term*. Specifically, it produces the term `(typespec-check 'ts term)`, where *ts* is the result of `(mfc-ts term mfc state :forcep nil :ttreep nil)` and `(typespec-check ts val)` is true when `val`'s actual type is in the type-set `ts`.
- `(:rw+ term alist obj equiv)` produces a term stating that `mfc-rw+` correctly rewrites *term* under substitution *alist* with objective *obj* under equivalence relation *equiv*. The form of the term produced is

  `(equiv term' rw)`

  where $term'$ is the new term formed by substituting *alist* into *term* and *rw* is the result of the call `(mfc-rw+ term alist obj equiv mfc state :forcep nil :ttreep nil)`. The *equiv* argument may also be `T`, meaning `IFF`, or `NIL`, meaning `EQUAL`.
- `(:rw term obj equiv)` is similar to the `:rw+` form above, but instead of `mfc-rw+` it supports `mfc-rw`, which takes no *alist* argument. Instead, `NIL` is used for the substitution.
- `(:ap term)` uses `mfc-ap` to derive a linear arithmetic contradiction indicating that *term* is false, and produces `(not term)` if that is successful, that is, if `(mfc-ap term mfc state :forcep nil)` returns true; otherwise it just produces `'T`.
- `(:relieve-hyp hyp alist rune target backptr)` uses `mfc-relieve-hyp` to attempt to prove that *hyp* holds under substitution *alist*, and produces the substitution of *alist* into *hyp* if successful, that is, if `(mfc-relieve-hyp hyp alist rune target backptr mfc state :forcep nil :ttreep nil)` returns true; otherwise, `'T`.

We have seen there are two forms of meta-extract hypotheses: `meta-extract-contextual-fact` and `meta-extract-global-fact+` (and its less general form, `meta-extract-global-fact`). We could have split each of these into several forms, resulting in eight forms for the eight kinds of values of `obj` listed above. However, in the next section we describe a utility that essentially generalizes the two supported forms, which eliminates the need for the user to think about the precise values of `obj`; it was convenient to generalize two supported forms rather than eight.

3 Using "meta-extract-user.lisp"

The community book "clause-processors/meta-extract-user.lisp" is designed to allow more convenient use of the meta-extract facility. The main contribution of this book is in the event-generating macro def-meta-extract. For a given pseudo-evaluator evl, def-meta-extract produces macros evl-meta-extract-contextual-facts and evl-meta-extract-global-facts that expand to meta-extract hypotheses where the obj argument is a call of a "bad-guy" (Skolem) function. This essentially universally quantifies the obj argument: informally, the term

$obj_0 =$ `(evl-meta-extract-contextual-bad-guy a mfc state)`

is some obj for which the formula

$\varphi =$ `(evl (meta-extract-contextual-fact obj mfc state) a)`

is false, if any such obj exists; so by asserting that φ is true for obj_0, we assert $(\forall\ obj)\ \varphi$. Therefore,

```
(evl (meta-extract-contextual-fact
       (evl-meta-extract-contextual-bad-guy a mfc state)
       mfc state)
     a)
```

implies that for any obj,

```
(evl (meta-extract-contextual-fact obj mfc state) a).
```

The def-meta-extract utility also proves several theorems about the pseudo-evaluator. They are proven by functional instantiation of similar theorems proved in "meta-extract-user.lisp" about a base evaluator that only supports the six functions typespec-check, if, equal, not, iff, and implies; this means that evl must also support at least these six functions for the utility to work.

The theorems proved by def-meta-extract obviate the need for the user to reason about the specifics of the definitions of meta-extract-contextual-fact and meta-extract-global-fact+ and the proper construction of their obj arguments, while still supporting all the facilities listed in Section 2.2. For example, this rule shows that (evl-meta-extract-global-facts) implies the correctness of meta-extract-formula (and makes no reference to the form of the obj argument to meta-extract-global-fact):

```
(defthm evl-meta-extract-formula
  (implies (and (evl-meta-extract-ev-global-facts)
                (equal (w st) (w state)))
           (evl (meta-extract-formula name st) a)))
```

This rule shows that (evl-meta-extract-contextual-facts a) implies the correctness of mfc-rw+ (specifically, when nil, meaning equal, is given as the equivalence relation argument):

```
(defthm evl-meta-extract-rw+-equal
   (implies (evl-meta-extract-contextual-facts a)
            (equal (evl (mfc-rw+
                         term alist obj nil
                         mfc state :forcep nil)
                        a)
                   (evl (sublis-var alist term) a))))
```

The above rule involves the system function `sublis-var`, which substitutes `alist` into `term` but reduces ground calls of primitive functions[5] to their values. In order to reason about `sublis-var`, the pseudo-evaluator should support all of the primitive functions that it treats specially. The community book "clause-processors/sublis-var-meaning.lisp" defines a pseudo-evaluator `cterm-ev` that supports exactly these functions and proves the following theorem that describes how `sublis-var` evaluates:

```
(defthm eval-of-sublis-var
  (implies (and (pseudo-termp x)
                (not (assoc nil alist)))
           (equal (cterm-ev (sublis-var alist x) a)
                  (cterm-ev x (append (cterm-ev-alist alist a) a)))))
```

To reason about `mfc-rw+`, `mfc-rw`, and `mfc-relieve-hyp`, whose meta-extract assumptions all involve `sublis-var`, one can define a pseudo-evaluator that supports both the functions required for `def-meta-extract` and for `sublis-var`. Community book "centaur/misc/context-rw.lisp", for example, defines a pseudo-evaluator supporting all these functions, uses `def-meta-extract`, and functionally instantiates the above theorem about `sublis-var` to allow it to reason about `mfc-rw+`.

4 Applications

4.1 GL Symbolic Interpreter

The GL framework for bit-level symbolic execution [8, 9] is an important tool for hardware verification efforts at Centaur Technology [2]. To prove a theorem, GL attempts to symbolically interpret the conclusion and render it into a Boolean formula which can be proved via a satisfiability check. Given a bit-level representation of each variable, the symbolic interpreter recursively computes a bit-level representation of each subterm. It expands function definitions as needed, down to certain primitive functions for which support is built in. Recent versions can also apply rewrite rules. It is implemented as a clause processor and uses meta-extract to look up function definitions and rewrite rules from the world and to evaluate ground terms using `magic-ev-fncall`.

GL was originally written before meta-extract, and used two tricks to replace its functionality. The complexity of these tricks reflects the utility of meta-extract, since it allows GL to now avoid these desperate measures.

- In order to justify the correctness of function definitions, GL would keep track of all definitions that were used, and return each definitional equation as an additional proof obligation from the clause processor. GL proof hints were orchestrated so as to use `:by` hints to attempt to cheaply discharge these obligations. GL did not yet use rewrite rules, but they could have been handled similarly.

- In order to apply functions to concrete arguments, each GL clause processor had an apply function that could call a fixed set of functions by name using a `case` statement. GL provided automation for defining new such clause processors so that users could build in their own set of functions.

[5] By "primitive functions" we mean built-in functions such as `binary-+` that do not have defining events — that is, those found in the ACL2 constant `*primitive-formals-and-guards*`.

4.2 Rewrite-bounds

The community book "centaur/misc/bound-rewriter.lisp" provides a tool for solving certain inequalities: it replaces subterms of an inequality with known bounds if those subterms are in monotonic positions. For example, the term $a - b$ monotonically decreases as b increases, so if we wish to prove $c < a - b$ and we know $B \geq b$, then it suffices to prove $c < a - B$. While this example would be easily handled by ACL2's linear arithmetic solver, there are problems that the bound rewriter can handle easily that overwhelm ACL2's nonlinear solver and cannot be solved with linear arithmetic – e.g.,

```
(implies (and (rationalp a) (rationalp b) (rationalp c)
              (<= 0 a) (<= 0 b) (<= 1 c)
              (<= a 10) (<= b 20) (<= c 30))
         (<= (+ (* a b c) (* a b) (* b c) (* a c))
             (+ (* 10 20 30) (* 10 20) (* 20 30) (* 10 30))))
```

This formula can't be solved with linear arithmetic, because it is not a linear problem. (If each product of variables is replaced by a fresh variable, the result is clearly not a theorem.) Moreover, the hint `:nonlinearp t` causes ACL2 to hang indefinitely. However, the hint

```
(rewrite-bounds ((<= a 10)
                 (<= b 20)
                 (<= c 30)))
```

solves it instantaneously by replacing upper-boundable occurrences of `a` by 10, `b` by 20, and `c` by 30. The same results are obtained — a quick proof using `rewrite-bounds` but an indefinite hang using nonlinear arithmetic — if the arithmetic expression on the last line of the theorem is replaced by its value, 7100.

The bound rewriter tool is implemented as a meta rule and uses meta-extract extensively. To determine which subterms are in monotonic positions, it uses type-set reasoning to determine the signs of subterms. For example, $a \cdot b$ increases as b increases if a is nonnegative and decreases as b increases if a is nonpositive; if we can't (weakly) determine the sign of a, then we can't replace b or any subterm with a bound. To determine whether a proposed bound of a subterm is (contextually) true, it uses `mfc-relieve-hyp` to show it by rewriting, and if that fails, `mfc-ap` to show it by linear arithmetic reasoning. The correctness of these uses of ACL2 reasoning utilities are justified by meta-extract-contextual-fact hypotheses.

4.3 Context-rw

A meta rule for context-sensitive rewriting, accomplishing something similar to Greve's "Nary" framework [4], is defined in "centaur/misc/context-rw.lisp" (see contextual-rewriting). Like Nary, it supports an analogue of congruence reasoning using contexts that are more general than equivalence relations. An example of its use is to allow, e.g.,

```
(logbitp 4 (logand (logior a b c (logapp 6 d e)) f g))
```

to be simplified to

```
(logbitp 4 (logand (logior a b c d) f g))
```

without defining a rewrite rule specifically for that case. The context rewriter uses certain theorems to justify rewriting subterms under new contexts, that is, with new calls wrapped around them. In this case we might have a rule that says that `(logbitp 4 ...)` induces a `(logand 16 ...)` context; this context can then be propagated through the `logand` and `logior` down to the `logapp` call, which simplifies under that context to just `d`, using a traditional rewrite rule such as the following:

```
(implies (and (syntaxp (and (quotep n) (quotep m)))
              (equal (logtail m n) 0))
         (equal (logand n (logapp m a b))
                (logand n a)))
```

When interpreted as a context rule, the following says that `logbitp` induces a `logand` context, by directing replacement of an instance of the right-hand side of the equality by the corresponding instance of the left-hand side.

```
(implies (syntaxp (quotep n))
         (equal (logbitp n (logand (ash 1 (nfix n)) m))
                (logbitp n m)))
```

And the following rule says that `logior` propagates such a `logand` context onto its second argument:

```
(implies (syntaxp (quotep n))
         (equal (logand n (logior a (logand n b)))
                (logand n (logior a b))))
```

A syntactic requirement for context rules is that the left- and right-hand sides must be identical except that some subterm of the LHS is replaced by a variable in the RHS (in this case `b`), and that is the only occurrence of that variable in the RHS. That variable corresponds to the subterm onto which the context will be propagated, and the corresponding subterm of the LHS reflects the context that will be propagated onto it. So the rule immediately above says:

> When `logior` occurs under a `(logand n ...)` context where `n` is a constant, propagate the `(logand n ...)` context onto the second argument of the `logior`.

Experienced ACL2 users might note that the roles of the left- and right-hand sides are essentially reversed in this usage; this is because often the reverse of a good context-propagation rule is also a good rewrite rule.

The context rewriter uses meta-extract in order to trust rewrite rules extracted from the world and results from `mfc-rw+` and `mfc-relieve-hyp`, which it uses, respectively, to simplify subterms under new contexts and to discharge hypotheses necessary for applying the context rules.

4.4 Others

A few other utilities from the community books use meta-extract solely to be able to extract a formula from the world (using `meta-extract-formula`) and assume it to be true. For example, "clause-processors/witness-cp.lisp" provides a framework for reasoning about quantification (see witness-cp); it uses `meta-extract-formula` to look up a stored fact showing that a term representing a universal quantification implies any instance of the quantified formula. A second, "clause-processors/just-expand.lisp", provides a clause processor and meta rule that force expansion of certain terms, somewhat similar to the `:expand` hint. A third, "clause-processors/replace-equalities.lisp", provides a tool for replacing known equalities in ways that the rewriter can't, e.g., replacing a variable with a term. For example, the following is not a valid rewrite rule because its left-hand side is a variable, but it could be a good replace-equalities rule:

```
(implies (matches pattern x)
         (equal x
                (patternsubst pattern (sigma pattern x))))
```

5 Conclusion

This paper explains meta-extract hypotheses and shows how they can be put to good use, either directly or by way of the `def-meta-extract` utility to obtain two simple, general meta-extract hypotheses. The implementation and logical justification for meta-extract are delicate, so it made sense to implement only the primitive notions of Section 2 in ACL2 and then introduce `def-meta-extract` in a book, with less trusted code as a result.

The meta-extract facility has been successfully used to create specialized proof tools that are admitted by ACL2 as fully verified metafunctions or clause-processors, including the GL bit-blasting framework, which is in daily use at Centaur Technology. We hope that this paper contributes to wider successful use of the meta-extract feature.

Acknowledgments

We thank J Moore for helpful discussions. We also thank the referees for useful feedback. This material is based upon work supported in part by DARPA under Contract No. FA8750-15-C-0007 (subcontract 15-C-0007-UT-Austin) and by ForrestHunt, Inc.

References

[1] ACL2 Community (accessed January, 2017): *ACL2+Books Documentation*. Available at `http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/index.html`.

[2] Anna Slobodova, Jared Davis, Sol Swords, and Warren A. Hunt, Jr. (2011): *A Flexible Formal Verification Framework for Industrial Scale Validation*. In: *9th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE), 2011*, pp. 89–97, doi:10.1109/MEMCOD.2011.5970515.

[3] R. S. Boyer & J S. Moore (1981): *Metafunctions: Proving Them Correct and Using Them Efficiently as New Proof Procedures*. In: *The Correctness Problem in Computer Science*, Academic Press, London.

[4] David Greve (2006): *Parameterized Congruences in ACL2*. In: *Proceedings of the Sixth International Workshop on the ACL2 Theorem Prover and Its Applications*, ACL2 '06, ACM, New York, NY, USA, pp. 28–34, doi:10.1145/1217975.1217981.

[5] W. A. Hunt, Jr., M. Kaufmann, R. B. Krug, J S. Moore & E. W. Smith (2005): *Meta Reasoning in ACL2*. In J. Hurd & T. Melham, editors: *18th International Conference on Theorem Proving in Higher Order Logics: TPHOLs 2005*, *Lecture Notes in Computer Science* 3603, Springer, pp. 163–178, doi:10.1007/11541868_11.

[6] Matt Kaufmann and J S. Moore (Accessed: 2016): *ACL2 home page*. (see URL `http://www.cs.utexas.edu/users/moore/acl2`).

[7] Matt Kaufmann, J S. Moore, Sandip Ray, and Erik Reeber (2009): *Integrating External Deduction Tools with ACL2*. *Journal of Applied Logic* 7(1), pp. 3 – 25, doi:10.1016/j.jal.2007.07.002. Special Issue: Empirically Successful Computerized Reasoning.

[8] Sol Swords (2010): *A Verified Framework for Symbolic Execution in the ACL2 Theorem Prover*. Ph.D. thesis, Department of Computer Sciences, The University of Texas at Austin. Available at `http://hdl.handle.net/2152/ETD-UT-2010-12-2210`.

[9] Sol Swords and Jared Davis (2011): *Bit-Blasting ACL2 Theorems*. In: *Proceedings 10th International Workshop on the ACL2 Theorem Prover and its Applications, ACL2 2011, Austin, Texas, USA, November 3-4, 2011.*, pp. 84–102, doi:10.4204/EPTCS.70.7.

A Versatile, Sound Tool for Simplifying Definitions

Alessandro Coglio
Kestrel Institute
3260 Hillview Avenue, Palo Alto, CA 94304, USA
coglio@kestrel.edu

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

Eric W. Smith
Kestrel Institute
3260 Hillview Avenue, Palo Alto, CA 94304, USA
eric.smith@kestrel.edu

We present a tool, `simplify-defun`, that transforms the definition of a given function into a simplified definition of a new function, providing a proof checked by ACL2 that the old and new functions are equivalent. When appropriate it also generates termination and guard proofs for the new function. We explain how the tool is engineered so that these proofs will succeed. Examples illustrate its utility, in particular for program transformation in synthesis and verification.

1 Introduction

We present a tool, `simplify-defun`, that transforms the definition of a given function into a simplified definition of a new function, providing a proof that the old and new functions are equivalent. When appropriate it also generates termination and guard proofs for the new function. Since the generated proofs are submitted to ACL2, `simplify-defun` need not be trusted: its soundness only depends on the soundness of ACL2. The new function is a 'simplified' version of the original function in much the same sense that ACL2 'simplifies' terms during proofs — via rewrite, type-set, forward chaining, and linear arithmetic rules.

`Simplify-defun` is one of the transformations of APT (Automated Program Transformations) [5], an ACL2 library of tools to transform programs and program specifications with a high degree of automation. APT can be used in program synthesis, to derive provably correct implementations from formal specifications via sequences of refinement steps carried out via transformations. APT can also be used in program analysis, to help verify existing programs, suitably embedded in the ACL2 logic, by raising their level of abstraction via transformations that are inverses of the ones used in stepwise program refinement. In the APT ecosystem, `simplify-defun` is useful for simplifying and optimizing definitions generated by other APT transformations (e.g., transformations that change data representation), as well as to carry out rewriting transformations via specific sets of rules (e.g., turning unbounded integer operations into bounded integer operations and vice versa, under suitable conditions). It can also be used to chain together and propagate sequences of transformations (e.g., rewriting a caller by replacing a callee with a new version, which may in turn be replaced with an even newer version, and so on).

The idea of using simplification rules to transform programs is not new [7, 2]. The contribution of the work described in this paper is the realization of that idea in ACL2, which involves techniques that are specific to this prover and environment and that leverage its existing capabilities and libraries. This paper could be viewed as a follow-up to an earlier paper on a related tool [6]. But that paper focused largely

Warren Hunt Jr. and Anna Slobodova (Eds.):
ACL2 Theorem Prover and its Applications
EPTCS 249, 2017, pp. 61–77, doi:10.4204/EPTCS.249.5

on usage, while here, we additionally focus both on interesting applications and on implementation.[1] Moreover, the new tool was implemented from scratch and improves on the old tool in several important ways, including the following:

- The new tool has significantly more options. Of special note is support for patterns that specify which subterms of the definition's body to simplify.
- The new tool has been subjected to a significantly larger variety of uses (approximately 300 uses as of mid-January 2017, not including artificial tests), illustrating its robustness and flexibility.
- The old tool generated events that were written to a file; calls of the new tool are event forms that can be placed in a book. (The old tool pre-dated make-event, which is used in the new tool.)
- The new tool takes advantage of the *expander* in community book misc/expander.lisp, rather than "rolling its own" simplification. Several recent improvements have been made to that book in the course of developing simplify-defun, which can benefit other users of the expander.

Simplify-defun may be applied to function symbols that have been defined using defun, possibly with mutual recursion — perhaps indirectly using a macro, for example, defund or define. An analogous tool, simplify-defun-sk, may be applied to function symbols defined with defun-sk, but we do not discuss it here.

Notice the underlining above. Throughout this paper, we underline hyperlinks to topics in the online documentation [1] for ACL2 and its books. We use ACL2 notation freely, abbreviating with an ellipsis (...) to indicate omitted text and sometimes modifying whitespace in displayed output.

The rest of this section introduces simplify-defun via some very simple examples that illustrate the essence of the tool. Section 2 presents some examples of the tool's use in program transformation; that section provides motivation for some of the features supported by simplify-defun. Section 3 summarizes options for controlling this tool. In Section 4 we discuss how simplify-defun circumvents ACL2's mercurial heuristics and sensitivity to theories so that proofs succeed reliably and automatically. We conclude in Section 5.

1.1 Simple Illustrative Examples

We start with a very simple example that captures much of the essence of simplify-defun.

```
(include-book "simplify-defun")
(defun f (x)
  (if (zp x) 0 (+ 1 1 (f (+ -1 x)))))
```

Next, we run simplify-defun to produce a new definition and a defthm with the formula shown below. **Note**: All defun forms generated by simplify-defun contain declare forms, which are generally omitted in this paper; also, whitespace is liberally edited.

```
ACL2 !>(simplify-defun f)
 (DEFUN F{1} (X)
   (IF (ZP X) 0 (+ 2 (F{1} (+ -1 X)))))
ACL2 !>:pf f-becomes-f{1}
(EQUAL (F X) (F{1} X))
ACL2 !>
```

[1] We thank the reviewers of the previous paper for suggesting further discussion of implementation. In a sense, we are finally getting around to that!

Several key aspects of a successful simplify-defun run are illustrated above:

- A new function symbol is defined, using the numbered-names utilities.
- The body of the new definition is a simplified version of the body of the original definition, but with the old function replaced by the new in recursive calls.
- A *'becomes'* theorem is proved, which states the equivalence of the old and new function.

This behavior of simplify-defun extends naturally to mutual recursion, in which case a new mutual-recursion event is generated together with 'becomes' theorems. Consider this definition.

```
(mutual-recursion
 (defun f1 (x) (if (consp x) (not (f2 (nth 0 x))) t))
 (defun f2 (x) (if (consp x) (f1 (nth 0 x)) t)))
```

The result presents no surprises when compared to our first example.

```
ACL2 !>(simplify-defun f1)
 (MUTUAL-RECURSION (DEFUN F1{1} (X)
                     (IF (CONSP X) (NOT (F2{1} (CAR X))) T))
                   (DEFUN F2{1} (X)
                     (IF (CONSP X) (F1{1} (CAR X)) T)))
ACL2 !>:pf f1-becomes-f1{1}
(EQUAL (F1 X) (F1{1} X))
ACL2 !>:pf f2-becomes-f2{1}
(EQUAL (F2 X) (F2{1} X))
ACL2 !>
```

Simplify-defun makes some attempt to preserve structure from the original definitions. For example, the body of the definition of f1 (above) is stored, as usual, as a translated term. As with most utilities that manipulate ACL2 terms, simplify-defun operates on translated terms. So the new defun event form could easily use the transformed body, shown here; notice that T is quoted.

```
ACL2 !>(body 'f1{1} nil (w state))
(IF (CONSP X) (NOT (F2{1} (CAR X))) 'T)
ACL2 !>
```

If we naively produced a user-level (untranslated) term from that body, the result would look quite different from the original definition's body.

```
ACL2 !>(untranslate (body 'f1{1} nil (w state)) nil (w state))
(OR (NOT (CONSP X)) (NOT (F2{1} (CAR X))))
ACL2 !>
```

Therefore, simplify-defun uses the directed-untranslate utility to untranslate the new (translated) body, heuristically using the old body (translated and untranslated) as a guide. This utility was implemented in support of simplify-defun, but it is of more general use (e.g., in other APT transformations).

There are many ways to control simplify-defun by using keyword arguments, as described in the next two sections. Here we show how to limit simplification to specified subterms. Consider:

```
(defun g (x y)
  (list (+ (car (cons x y)) 3)
        (* (car (cons y y)) 4)
        (* (car (cons x y)) 5)))
```

The `:simplify-body` keyword option below specifies simplification of any occurrence of `(car (cons x y))` that is the first argument of a call to `*`. The wrapper `:@` indicates the simplification site, and the underscore (`_`) matches anything. Notice that, in the result, only the indicated call is simplified.

```
ACL2 !>(simplify-defun g :simplify-body (* (:@ (car (cons x y))) _))
 (DEFUN G{1} (X Y)
   (LIST (+ (CAR (CONS X Y)) 3)
         (* (CAR (CONS Y Y)) 4)
         (* X 5)))
ACL2 !>
```

2 Some Applications

This section presents some practical examples of use of `simplify-defun` in program transformation. They use some keyword options, which are described in Section 3 but we hope are self-explanatory here. Not shown here are hints generated by `simplify-defun` to automate proofs, for example by reusing previous guards, measures, and guard and termination theorems; this is covered briefly in subsequent sections.

2.1 Combining a Filter with a Doubly-Recursive Producer

This example shows how `simplify-defun` is used to apply rewrite rules, to improve the results of other transformations, and to chain together previous transformation steps. The main function `f` below produces all pairs of items from `x` and `y` and then filters the result to keep only the "good" pairs.

```
(defun pair-with-all (item lst) ;; pair ITEM with all elements of LST
  (if (endp lst)
      nil
    (cons (cons item (car lst))
          (pair-with-all item (cdr lst)))))

(defun all-pairs (x y) ;; make all pairs of items from X and Y
  (if (endp x)
      nil
    (append (pair-with-all (car x) y)
            (all-pairs (cdr x) y))))

(defstub good-pair-p (pair) t) ;; just a place holder

(defun keep-good-pairs (pairs)
  (if (endp pairs)
      nil
```

```
    (if (good-pair-p (car pairs))
        (cons (car pairs) (keep-good-pairs (cdr pairs)))
      (keep-good-pairs (cdr pairs)))))

(defun f (x y) (keep-good-pairs (all-pairs x y)))
```

We wish to make `f` more efficient; it should refrain from ever adding non-good pairs to the result, rather than filtering them out later. `F`'s body is `(keep-good-pairs (all-pairs x y))`, which we can improve by "pushing" `keep-good-pairs` into the `if`-branches of `all-pairs`, using APT's `wrap-output` transformation (not described here). `Wrap-output` produces a function and a theorem.

```
(DEFUN ALL-GOOD-PAIRS (X Y) ; generated by wrap-output
  (IF (ENDP X)
      (KEEP-GOOD-PAIRS NIL)
      (KEEP-GOOD-PAIRS (APPEND (PAIR-WITH-ALL (CAR X) Y)
                               (ALL-PAIRS (CDR X) Y)))))

(DEFTHM RULE1 ; generated by wrap-output
  (EQUAL (KEEP-GOOD-PAIRS (ALL-PAIRS X Y))
         (ALL-GOOD-PAIRS X Y)))
```

Below, we will apply `rule1` to simplify `f`. But first we will further transform `all-good-pairs`. It can be simplified in three ways. First, `(keep-good-pairs nil)` can be evaluated. Second, we can push the call to `keep-good-pairs` over the `append` using this rule.

```
(defthm keep-good-pairs-of-append
  (equal (keep-good-pairs (append x y))
         (append (keep-good-pairs x) (keep-good-pairs y))))
```

Third, note that `all-good-pairs`, despite being a transformed version of `all-pairs`, is not recursive (it calls the old function `all-pairs`), but we want it to be recursive. After `keep-good-pairs` is pushed over the `append`, it will be composed with the call of `all-pairs`, which is the exact pattern that `rule1` can rewrite to a call to `all-good-pairs`. `Simplify-defun` applies these simplifications.

```
ACL2 !>(simplify-defun all-good-pairs)
 (DEFUN ALL-GOOD-PAIRS{1} (X Y)
   (IF (ENDP X)
       NIL
       (APPEND (KEEP-GOOD-PAIRS (PAIR-WITH-ALL (CAR X) Y))
               (ALL-GOOD-PAIRS{1} (CDR X) Y))))
ACL2 !>
```

Note that the new function is recursive. This is because `rule1` introduced a call to `all-good-pairs`, which `simplify-defun` then renamed to `all-good-pairs{1}` (it always renames recursive calls). We have made some progress pushing `keep-good-pairs` closer to where the pairs are created. Now, `all-good-pairs{1}` can be further simplified. Observe that its body contains composed calls of `keep-good-pairs` and `pair-with-all`. We can optimize this term using APT's `producer-consumer` transformation (not described here) to combine the creation of the pairs with the filtering of good pairs. As usual, a new function and a theorem are produced.

```
(DEFUN PAIR-WITH-ALL-AND-FILTER (ITEM LST) ; generated by producer-consumer
  (IF (ENDP LST)
      NIL
      (IF (GOOD-PAIR-P (CONS ITEM (CAR LST)))
          (CONS (CONS ITEM (CAR LST))
                (PAIR-WITH-ALL-AND-FILTER ITEM (CDR LST)))
          (PAIR-WITH-ALL-AND-FILTER ITEM (CDR LST)))))

(DEFTHM RULE2 ; generated by producer-consumer
  (EQUAL (KEEP-GOOD-PAIRS (PAIR-WITH-ALL ITEM LST))
         (PAIR-WITH-ALL-AND-FILTER ITEM LST)))
```

`Pair-with-all-and-filter` immediately discards non-good pairs, saving work compared to filtering them out later. Now `simplify-defun` can change `all-good-pairs{1}` by applying `rule2`.

```
ACL2 !>(simplify-defun all-good-pairs{1})
 (DEFUN ALL-GOOD-PAIRS{2} (X Y)
   (IF (ENDP X)
       NIL
       (APPEND (PAIR-WITH-ALL-AND-FILTER (CAR X) Y)
               (ALL-GOOD-PAIRS{2} (CDR X) Y))))
ACL2 !>
```

Finally, we apply `simplify-defun` to transform `f` by applying all of the preceding rewrites in succession, introducing `all-good-pairs`, which is in turn replaced with `all-good-pairs{1}` and then `all-good-pairs{2}`.

```
ACL2 !>(simplify-defun f :new-name f-fast)
 (DEFUN F-FAST (X Y)
   (ALL-GOOD-PAIRS{2} X Y))
ACL2 !>:pf f-becomes-f-fast
(EQUAL (F X Y) (F-FAST X Y))
```

This builds a fast version of `f` and a theorem proving it equal to `f`.

2.2 Converting between Unbounded and Bounded Integer Operations

Popular programming languages like C and Java typically use bounded integer types and operations, while requirements specifications typically use unbounded integer types and operations. Thus, synthesizing a C or Java program from a specification, or proving that a C or Java program complies with a specification, often involves showing that unbounded and bounded integers are "equivalent" under the preconditions stated by the specification.

Consider this Java implementation of Bresenham's line drawing algorithm [3] for the first octant.[2]

```
// draw a line from (0, 0) to (a, b), where 0 <= b <= a <= 1,000,000:
static void drawLine(int a, int b) {
```

[2]This algorithm computes a best-fit discrete line using only integer operations. Understanding the algorithm is not necessary for the purpose of this `simplify-defun` example.

```
    int x = 0, y = 0, d = 2 * b - a;
    while (x <= a) {
        drawPoint(x, y); // details unimportant
        x++;
        if (d >= 0) { y++; d += 2 * (b - a); }
        else { d += 2 * b; }
    }
}
```

Assuming the screen width and height are less than 1,000,000 pixels, none of the two's complement 32-bit integer operations in the Java method wrap around. So they could be replaced with corresponding unbounded integer operations, as shown below. This replacement raises the level of abstraction and helps verify the functional correctness of the method.

The Java code above can be represented as shown below in ACL2 (see the paper's supporting materials for full details), where:

- `Int32p` recognizes some representation of Java's two's complement 32-bit integers, whose details are unimportant.
- `Int32` converts an ACL2 integer in $[-2^{31}, 2^{31})$ (i.e., an `x` such that `(signed-byte-p 32 x)` holds) to the corresponding representation in `int32p`.
- `Int` converts a representation in `int32p` to the corresponding ACL2 integer in $[-2^{31}, 2^{31})$.
- `Add32`, `sub32`, and `mul32` represent Java's two's complement 32-bit addition, subtraction, and multiplication operations.
- `Lte32` and `gte32` represent Java's two's complement 32-bit less-than-or-equal-to and greater-than-or-equal-to operations.
- `Drawline-loop` represents the loop as a state transformer, whose state consists of `a`, `b`, `x`, `y`, `d`, and the screen. This function is "defined" only where the loop invariant holds. The guard verification of this function implies the preservation of the loop invariant.
- `Drawline` represents the method as a function that maps `a`, `b`, and the current screen to an updated screen. This function is "defined"only where the precondition holds. The guard verification of this function implies the establishment of the loop invariant.

The Axe tool [8] can automatically generate a representation similar to this one from Java (byte)code, with some additional input from the user (e.g., part of the loop invariant).

```
(defun drawpoint (x y screen)
  (declare (xargs :guard (and (int32p x) (int32p y))))
  ...) ; returns updated screen, details unimportant

(defun precond (a b) ; precondition of the method
  (declare (xargs :guard t))
  (and (int32p a) ; Java type of a
       (int32p b) ; Java type of b
       (<= 0 (int b))
       (<= (int b) (int a))
```

```
       (<= (int a) 1000000)))

(defun invar (a b x y d) ; loop invariant of the method
  (declare (xargs :guard t))
  (and (precond a b)
       (int32p x) ; Java type of x
       (int32p y) ; Java type of y
       (int32p d) ; Java type of d
       ...)) ; conditions on x, y, and d, details unimportant

(defun drawline-loop (a b x y d screen) ; loop of the method
  (declare (xargs :guard (invar a b x y d)
                  ...)) ; measure and (guard) hints, details unimportant
  (if (invar a b x y d)
      (if (not (lte32 x a))
          screen ; exit loop
        (drawline-loop a b
                       (add32 x (int32 1))
                       (if (gte32 d (int32 0))
                           (add32 y (int32 1))
                         y)
                       (if (gte32 d (int32 0))
                           (add32 d (mul32 (int32 2) (sub32 b a)))
                         (add32 d (mul32 (int32 2) b)))
                       (drawpoint x y screen)))
    :undefined))

(defun drawline (a b screen) ; method
  (declare (xargs :guard (precond a b)
                  ...)) ; guard hints, details unimportant
  (if (precond a b)
      (drawline-loop a b
                     (int32 0) ; x
                     (int32 0) ; y
                     (sub32 (mul32 (int32 2) b) a) ; d
                     screen)
    :undefined))
```

The following rewrite rules are disabled because their right-hand sides are not generally "simpler" or "better" than their left-hand sides. But when passed to the `:enable` option of `simplify-defun`, which instructs the tool to use these rules in the expander, they systematically replace bounded integer operations with their unbounded counterparts.

```
(defthmd add32-to-+  (equal (add32 x y) (int32 (+ (int x) (int y)))))
(defthmd sub32-to--  (equal (sub32 x y) (int32 (- (int x) (int y)))))
(defthmd mul32-to--  (equal (mul32 x y) (int32 (* (int x) (int y)))))
(defthmd lte32-to-<= (equal (lte32 x y) (<= (int x) (int y))))
```

```
(defthmd gte32-to-<= (equal (gte32 x y) (>= (int x) (int y))))
```

Since these rewrite rules are unconditional, the replacement always occurs, but subterms of the form `(int (int32 ...))` are generated. For instance, the term `(add32 d (mul32 (int32 2) b))` above becomes `(int32 (+ (int d) (int (int32 (* (int (int32 2)) (int b))))))`. These `(int (int32 ...))` terms can be simplified via the following conditional rewrite rule (which the expander in `simplify-defun` uses by default, since it is an enabled rule). Relieving the hypotheses of this rewrite rule's applicable instances amounts to showing that each bounded integer operation does not wrap around in the expressions under consideration.

```
(defthm int-of-int32
  (implies (signed-byte-p 32 x)
           (equal (int (int32 x)) x)))
```

Applying `simplify-defun` to `drawline-loop` and `drawline` yields the desired results. In this case, the `int-of-int32` hypotheses are automatically relieved. These uses of `simplify-defun` show a practical use of the pattern feature: `invar` and `precond` must be enabled to relieve the `int-of-int32` hypotheses, but we want the generated function to keep them unopened; so we use a pattern that limits the simplification to the true branches of the `if`s. The 'becomes' theorem generated by the first `simplify-defun` is used by the second to have `drawline{1}` call `drawline-loop{1}` instead of `drawline-loop`; this is another example of propagating transformations.

```
ACL2 !> (simplify-defun drawline-loop
                        :simplify-body (if _ @ _)
                        :enable (add32-to-+ ... gte32-to->=))
 (DEFUN DRAWLINE-LOOP{1} (A B X Y D SCREEN)
   (DECLARE ...)
   (IF (INVAR A B X Y D)
       (IF (NOT (< (INT A) (INT X)))
           (DRAWLINE-LOOP{1} A B
                             (INT32 (+ 1 (INT X)))
                             (IF (< (INT D) 0)
                                 Y
                               (INT32 (+ 1 (INT Y))))
                             (IF (< (INT D) 0)
                                 (INT32 (+ (INT D) (* 2 (INT B))))
                               (INT32 (+ (INT D)
                                         (- (* 2 (INT A)))
                                         (* 2 (INT B)))))
                             (DRAWPOINT X Y SCREEN))
     SCREEN)
    :UNDEFINED))
ACL2 !> (simplify-defun drawline
                        :simplify-body (if _ @ _)
                        :enable (add32-to-+ ... gte32-to->=))
 (DEFUN DRAWLINE{1} (A B SCREEN)
   (DECLARE ...)
   (IF (PRECOND A B)
```

```
       (DRAWLINE-LOOP{1} A B
                         (INT32 0)
                         (INT32 0)
                         (INT32 (+ (- (INT A)) (* 2 (INT B))))
                         SCREEN)
     :UNDEFINED))
ACL2 !>
```

The resulting expressions have the `int` conversion at the variable leaves and the `int32` conversion at the roots. APT's isomorphic data transformation (not discussed here) can eliminate them by changing the data representation of the functions' arguments, generating functions that no longer deal with bounded integers. The resulting functions are more easily proved to satisfy the high-level functional specification of the algorithm, namely that it produces a best-fit discrete line (this proof is not discussed here).

The bounded-to-unbounded operation rewriting technique shown here, followed by the isomorphic data transformation mentioned above, should have general applicability. Proving that bounded integer operations do not wrap around may be arbitrarily hard: when the `int-of-int32` hypotheses cannot be relieved automatically, the user may have to prove lemmas to help `simplify-defun`. When a Java computation is supposed to wrap around (e.g., when calculating a hash), the specification must explicitly say that, and slightly different rewrite rules may be needed. When synthesizing code (as opposed to analyzing code as in this example), it should be possible to use a similar technique with rewrite rules that turn unbounded integer operations into their bounded counterparts, "inverses" of the rules `add32-to-+`, `sub32-to--`, etc.

3 Options

This section very briefly summarizes the keyword arguments of `simplify-defun`. Here we assume that the given function's definition is not mutually recursive; for that case and other details, see the XDOC topic for simplify-defun, provided by the supporting materials.

Assumptions. A list of assumptions under which the body is simplified can be specified by the `:assumptions` keyword. Or, keyword `:hyp-fn` can specify the assumptions using a function symbol.

Controlling the Result. By default, the new function symbol is disabled if and only if the input function symbol is disabled. However, that default can be overridden with keyword `:function-disabled`. Similarly, the measure, guard verification, and non-executability status come from the old function symbol but can be overridden by keywords `:measure`, `:verify-guards`, and `:non-executable`, respectively. The 'becomes' theorem is enabled by default, but this can be overridden by keyword `:theorem-disabled`. Keywords can also specify the names for the new function symbol (`:new-name`) and 'becomes' theorem (`:theorem-name`). The new function body is produced, by default, using the `directed-untranslate` utility (see Section 1); but keyword `:untranslate` can specify to use the ordinary untranslate operation or even to leave the new body in translated form. Finally, by default the new function produces results equal to the old; however, an equivalence relation between the old and new results can be specified with keyword `:equiv`, which is used in the statement of the 'becomes' theorem.

Specifying Theories. The `:theory` keyword specifies the theory to be used when simplifying the definition; alternatively, `:disable` and `:enable` keywords can be used for this purpose. Keyword `:expand` can be used with any (or none) of these to specify terms to expand, as with ordinary `:expand` hints for the prover. Similarly, there are keywords `:assumption-theory`, `:assumption-disable`, and `:assumption-enable` for controlling the theory used when proving that assumptions are preserved

by recursive calls (an issue discussed in Section 4.3). There are also keywords `:measure-hints` and `:guard-hints` with the obvious meanings.

Specifying Simplification. By default, `simplify-defun` attempts to simplify the body of the given function symbol, but not its guard or measure. Keywords `:simplify-body`, `:simplify-guard`, and `:simplify-measure` can override that default behavior. By default, `simplify-defun` fails if it attempts to simplify the body but fails to do so, though there is no such requirement for the guard or measure; keyword `:must-simplify` can override those defaults.

Output Options. The keyword `:show-only` causes `simplify-defun` not to change the world, but instead to show how it expands into primitive events (see Section 4). If `:show-only` is `nil` (the default), then by default, the new definition is printed when `simplify-defun` is successful; keyword `:print-def` can suppress that printing. Finally, a `:verbose` option can provide extra output.

4 Implementation

`Simplify-defun` is designed to apply the *expander*, specifically, function `tool2-fn` in community book `misc/expander.lisp`, which provides an interface to the ACL2 rewriter in a context based on the use of forward-chaining and linear arithmetic. The goal is to simplify the definition as specified and to arrange that all resulting proofs succeed fully automatically and quickly.

This section discusses how `simplify-defun` achieves this goal, with a few (probably infrequent) exceptions in the case of proofs for guards, termination, or (discussed below) that assumptions are preserved by recursive calls. First, we explain the full form generated by a call of `simplify-defun`, which we call its *expansion*; this form carefully orchestrates the proofs. Then we dive deeper by exploring the proof of the 'becomes' theorem, focusing on the use of functional instantiation. Next we see how the expansion is modified when assumptions are used. Finally we provide a few brief implementation notes.

One can experiment using the supporting materials: see the book `simplify-defun-tests.lisp`; or simply include the book `simplify-defun`, define a function `f`, and then evaluate `(simplify-defun f :show-only t)`, perhaps adding options, to see the expansion.

The implementation takes advantage of many features offered by ACL2 for system building (beyond its prover engine), including for example `encapsulate`, `make-event`, `with-output`. We say a bit more about this at the end of the section.

4.1 The `Simplify-defun` Expansion

We illustrate the expansion generated by `simplify-defun` using the following definition, which is the first example of Section 1 but with a guard added.

```
(defun f (x)
 (declare (xargs :guard (natp x)))
 (if (zp x) 0 (+ 1 1 (f (+ -1 x)))))
```

The expansion is an `encapsulate` event. We can see the expansion by evaluating the form `(simplify-defun f :show-only t)`, which includes the indicated four sections to be discussed below.

```
(encapsulate nil
 prelude
 new defun form
 (local (progn local events))
 'becomes' theorem)
```

4.1.1 Prelude

The prelude is mostly independent of `f`, but the name of `f` is supplied to `install-not-normalized`.

```
(SET-INHIBIT-WARNINGS "theory")
(SET-IGNORE-OK T)
(SET-IRRELEVANT-FORMALS-OK T)
(LOCAL (INSTALL-NOT-NORMALIZED F))
(LOCAL (SET-DEFAULT-HINTS NIL))
(LOCAL (SET-OVERRIDE-HINTS NIL))
```

The first form above avoids warnings due to the use of small theories for directing the prover. The next two forms allow simplification to make some formals unused or irrelevant in the new definition. The use of `install-not-normalized` is a bit subtle perhaps, but not complicated: by default, ACL2 stores a simplified, or *normalized*, body for a function symbol, but `simplify-defun` is intended to generate a definition based on the body *b* of the old definition as it was submitted, not on the normalization of *b*. Using `install-not-normalized` arranges that `:expand` hints for `f` will use the unnormalized body. This supports the proofs generated by `simplify-defun`, by supporting reasoning about the unnormalized bodies of both the old and new functions. Finally, the last two forms guarantee that the global environment will not sabotage the proof.

Space does not permit discussion of the handling of `mutual-recursion`. We mention here only that the form `(SET-BOGUS-MUTUAL-RECURSION-OK T)` is then added to the prelude, in case some of the (mutual) recursion disappears with simplification.

4.1.2 New Defun Form

The following new definition has the same simplified body as for the corresponding example in Section 1. As before, the new body is produced by running the expander on the unnormalized body of `f`.

```
(DEFUN F{1} (X)
  (DECLARE (XARGS :NORMALIZE NIL
                  :GUARD (NATP X)
                  :MEASURE (ACL2-COUNT X)
                  :VERIFY-GUARDS T
                  :GUARD-HINTS (("Goal" :USE (:GUARD-THEOREM F)) ...)
                  :HINTS (("Goal" :USE (:TERMINATION-THEOREM F)) ...)))
  (IF (ZP X) 0 (+ 2 (F{1} (+ -1 X)))))
```

The guard and measure are (by default) inherited from `f`. Since `f` is guard-verified, then by default, so is the new function. The uses of `:GUARD-THEOREM` and `:TERMINATION-THEOREM` are designed to make the guard and termination proofs automatic and fast in most cases, and were implemented in support of the APT project.

4.1.3 Local Events

The local events are how `simplify-defun` carefully arranges for proofs to succeed automatically and fast, in most cases. The first local event defines a theory to be used when proving equality of the body of `f` with a simplified version. It is the union of the runes reported by the expander when simplifying

the body, with the set of all :congruence and :equivalence runes, since these are not tracked by the ACL2 rewriter.[3]

```
(MAKE-EVENT (LET ((THY (UNION-EQUAL '((:REWRITE FOLD-CONSTS-IN-+)
                                      (:EXECUTABLE-COUNTERPART BINARY-+)
                                      (:DEFINITION SYNP))
                                    (CONGRUENCE-THEORY (W STATE)))))
                 (LIST 'DEFCONST '*F-RUNES* (LIST 'QUOTE THY))))
```

The second local event proves the equality of the body with its simplified version. Notice that the latter is still in terms of f; the new function symbol, f{1}, will be introduced later. The proof-builder :instructions are carefully generated to guarantee that the proof succeeds; in this simple example, they first put the proof-builder in the smallest theory that should suffice (for efficiency), then simplify the old body (the first argument of the equal call), and then prove the resulting equality (which at that point should be the equality of two identical terms).

```
(DEFTHM F-BEFORE-VS-AFTER-0
  (EQUAL (IF (ZP X) 0 (+ 1 1 (F (+ -1 X))))
         (IF (ZP X) 0 (+ 2 (F (+ -1 X)))))
  :INSTRUCTIONS ...
  :RULE-CLASSES NIL)
```

The third local event is as follows.

```
(COPY-DEF F{1}
          :HYPS-FN NIL
          :HYPS-PRESERVED-THM-NAMES NIL
          :EQUIV EQUAL)
```

This macro call introduces a constrained function symbol, f{1}-copy, whose constraint results from the definitional axiom for f{1} by replacing f{1} with f{1}-copy. It also proves the two functions equivalent using a trivial induction in a tiny theory, to make the proof reliable and fast.

```
(DEFTHM F{1}-COPY-DEF
  (EQUAL (F{1}-COPY X)
         (IF (ZP X)
             '0
             (BINARY-+ '2 (F{1}-COPY (BINARY-+ '-1 X)))))
  :HINTS ...
  :RULE-CLASSES ((:DEFINITION :INSTALL-BODY T
                              :CLIQUE (F{1}-COPY)
                              :CONTROLLER-ALIST ((F{1}-COPY T)))))

(LOCAL (IN-THEORY '((:INDUCTION F{1}) F{1}-COPY-DEF (:DEFINITION F{1}))))
```

[3] We originally generated a form (defconst *f-runes* ...) that listed all runes from the union-equal call; but the congruence theory made that list long and distracting when viewing the expansion with :show-only t. Note that the congruence theory includes :equivalence runes, which after all represent congruence rules for diving into calls of equivalence relations.

```
(DEFTHM F{1}-IS-F{1}-COPY
  (EQUAL (F{1} X) (F{1}-COPY X))
  :HINTS (("Goal" :INDUCT (F{1} X)))
  :RULE-CLASSES NIL)
```

The last local event is the lemma for proving the 'becomes' theorem. In Section 4.2 below we explore this use of functional instantiation.

```
(DEFTHM F-BECOMES-F{1}-LEMMA
  (EQUAL (F{1} X) (F X))
  :HINTS (("Goal"
           :BY (:FUNCTIONAL-INSTANCE F{1}-IS-F{1}-COPY (F{1}-COPY F))
           :IN-THEORY (UNION-THEORIES (CONGRUENCE-THEORY WORLD)
                                      (THEORY 'MINIMAL-THEORY)))
          '(:USE (F-BEFORE-VS-AFTER-0 F$NOT-NORMALIZED))))
```

4.1.4 'Becomes' Theorem

The 'becomes' theorem in this example states the same theorem as its lemma above (though we will see in Section 4.3 that this is not always be the case). The `:in-theory` hint serves to keep the ACL2 rewriter from bogging down during the proof.

```
(DEFTHM F-BECOMES-F{1}
  (EQUAL (F X) (F{1} X))
  :HINTS (("Goal" :USE F-BECOMES-F{1}-LEMMA :IN-THEORY NIL)))
```

4.2 Proving the 'Becomes' Theorem

Next we see how functional instantiation is used in proving the 'becomes' theorem above, or more precisely, its local lemma. Recall that above, a `:by` hint is used that replaces `f{1}-copy` by `f` in the lemma `f{1}-is-f{1}-copy`, `(EQUAL (F{1} X) (F{1}-COPY X))`, to prove: `(EQUAL (F X) (F{1} X))`. That substitution works perfectly (modulo commuting the equality, which the `:by` hint tolerates), but it requires proving the following property, which states that `f` satisfies the constraint for `f{1}-copy`.

```
(EQUAL (f X)
       (IF (ZP X)
           '0
           (BINARY-+ '2 (f (BINARY-+ '-1 X)))))
```

But the right-hand side is exactly what was produced by applying the expander to the body of `f`. If we look at the `:hints` in `f-becomes-f{1}-lemma` above, we see that after using functional instantiation (with the `:by` hint), a computed hint completes the proof by using two facts: `f-before-vs-after-0` (the second local lemma above), which equates the two bodies; and `f$not-normalized`, which equates `f` with its unnormalized (i.e., user-supplied) body. The latter was created in the prelude (Section 4.1.1) by the form `(install-not-normalized f)`. Notice that no proof by induction was performed for the 'becomes' theorem (or its local lemma), even though it is inherently an inductive fact stating the equivalence of recursive functions `f` and `f{1}`. We are essentially taking advantage of the trivial induction already performed in the proof of the lemma being functionally instantiated, `f{1}-is-f{1}-copy`.

4.3 Assumptions

The following trivial example shows how assumptions change the `simplify-defun` expansion.

```
(defun foo (x)
    (declare (xargs :guard (true-listp x)))
    (if (consp x)
        (foo (cdr x))
      x))
```

Under the `:assumption` of the guard, `(true-listp x)`, the variable `x` in the body is simplified to the constant `nil`, using its context `(not (consp x))`.

```
ACL2 !>(simplify-defun foo :assumptions :guard :show-only t)
(ENCAPSULATE NIL
 prelude (as before)
 ; new defun form:
 (DEFUN FOO{1} (X)
   (IF (CONSP X) (FOO{1} (CDR X)) NIL))
 (LOCAL (PROGN local events))  ; discussed below
 ; 'becomes' theorem:
 (DEFTHM FOO-BECOMES-FOO{1}
   (IMPLIES (TRUE-LISTP X)
            (EQUAL (FOO X) (FOO{1} X)))
   :HINTS ...)
ACL2 !>
```

This time, the 'becomes' theorem has a hypothesis provided by the `:assumptions`.

We next discuss some differences in the local events generated when there are assumptions. A new local event defines a function for the assumptions, so that when the assumptions are complicated, disabling that function can hide complexity from the rewriter.

```
(DEFUN FOO-HYPS (X) (TRUE-LISTP X))
```

In order to prove equivalence of the old and new functions, which involves a proof by induction at some point, it is necessary to reason that the assumptions are preserved by recursive calls. The following local lemma is generated for that purpose.

```
(DEFTHM FOO-HYPS-PRESERVED-FOR-FOO
  (IMPLIES (AND (FOO-HYPS X) (CONSP X))
           (FOO-HYPS (CDR X)))
  :HINTS (("Goal" :IN-THEORY (DISABLE* FOO (:E FOO) (:T FOO))
                  :EXPAND ((:FREE (X) (FOO-HYPS X)))
                  :USE (:GUARD-THEOREM FOO)))
  :RULE-CLASSES NIL)
```

In many cases the proof of such a lemma will be automatic. Otherwise, one can first define `foo-hyps` and prove this lemma before running `simplify-defun`.

There are some tricky wrinkles in the presence of assumptions that we do not discuss here, in particular how they can affect the use of `copy-def` (discussed above in Section 4.1.3). Some relevant discussion may be found in the "Essay on the Implementation of Simplify-defun" in the file `simplify-defun.lisp`.

4.4 Implementation Notes

The discussion above explains the expansion from a call of `simplify-defun`. Here we discuss at a high level how such forms are generated. Consider the first example from the introduction, below, and let us see its single-step macroexpansion.

```
(defun f (x)
  (if (zp x) 0 (+ 1 1 (f (+ -1 x)))))

ACL2 !>:trans1 (simplify-defun f)
 (WITH-OUTPUT :GAG-MODE NIL :OFF :ALL :ON ERROR
   (MAKE-EVENT (SIMPLIFY-DEFUN-FN 'F 'NIL 'NIL ':NONE ... STATE)))
ACL2 !>
```

This use of `with-output` prevents output unless there is an error. The `make-event` call instructs `simplify-defun-fn` to produce an event of the form `(progn` *E A* `(value-triple 'D))`, where *E* is the expansion, *A* is a `table` event provided to support redundancy for `simplify-defun`, and *D* is the new definition (which is thus printed to the terminal). `Simplify-defun-fn` first calls the expander to produce a simplified body, which it then uses to create *D* and *E*. For details see `simplify-defun.lisp` in the supporting materials, which we hope is accessible to those having a little familiarity with ACL2 system programming (see for example system-utilities and programming-with-state). These details help proofs to succeed efficiently, in particular by generating suitable hints, including small theories. Another detail is that if the old definition specifies ruler-extenders other than the default, then these are carried over to the new definition.

5 Conclusion

The use of simplification, particularly rewrite rules, is an old and important idea in program transformation. `Simplify-defun` realizes this idea in ACL2, by leveraging the prover's existing proof procedures, libraries, and environment. It is one of the transformations of the APT tool suite for transforming programs and program specifications, useful for both synthesis and verification. While `simplify-defun` is appropriate for equivalence-preserving refinements, other APT transformations are appropriate for other kinds of refinement. For instance, specifications that allow more than one implementation (see [4, Section 2] for an example) can be refined via APT's narrowing transformation (not discussed here).

We have used `simplify-defun` quite extensively in program derivations, demonstrating its robustness and utility. This paper describes not only the general usage of `simplify-defun`, but also ACL2-specific techniques used to implement it. It also shows some non-trivial examples that illustrate the tool's utility.

The tool continues to evolve. Developments since the drafting of this paper include: delaying guard verification; and expanding all `LET` expressions, but reconstructing them with `directed-untranslate`. These enhancements and others will be incorporated into `simplify-defun` when we add it, soon, to the community books, located somewhere under `books/kestrel`.

Acknowledgments

This material is based upon work supported in part by DARPA under Contract No. FA8750-15-C-0007. We thank the referees, all of whom gave very helpful feedback that we incorporated.

References

[1] ACL2 Community (accessed December, 2016): *ACL2+Books Documentation*. See URL `http://www.cs.utexas.edu/~moore/acl2/manuals/current/manual?topic=ACL2____ACL2`.

[2] Richard S. Bird & Oege de Moor (1997): *Algebra of programming*. Prentice Hall International series in computer science, Prentice Hall.

[3] Jack Bresenham (1965): *Algorithm for Computer Control of a Digital Plotter*. *IBM Systems Journal* 4(1), pp. 25–30, doi:10.1147/sj.41.0025.

[4] Alessandro Coglio (2015): *Second-Order Functions and Theorems in ACL2*. In: *Proceedings of the Thirteenth International Workshop on the ACL2 Theorem Prover and its Applications*, doi:10.1145/1637837.1637839.

[5] Kestrel Institute & University of Texas at Austin (accessed January, 2017): *APT (Automated Program Transformations)*. `http://www.kestrel.edu/home/projects/apt`.

[6] Matt Kaufmann (2003): *A Tool for Simplifying Files of ACL2 Definitions*. In: *Proceedings of the Fourth International Workshop on the ACL2 Theorem Prover and its Applications, ACL2 2003, Boulder, Colorado, USA, July 13-14, 2003*.

[7] Douglas R. Smith (1990): *KIDS: A Semiautomatic Program Development System*. *IEEE Trans. Software Eng.* 16(9), pp. 1024–1043, doi:10.1109/32.58788.

[8] Eric W. Smith (2011): *Axe: An Automated Formal Equivalence Checking Tool for Programs*. Ph.D. dissertation, Stanford University.

Proof Reduction of Fair Stuttering Refinement of Asynchronous Systems and Applications

Rob Sumners
Centaur Technology
rsumners@centtech.com

We present a series of definitions and theorems demonstrating how to reduce the requirements for proving system refinements ensuring containment of fair stuttering runs. A primary result of the work is the ability to reduce the requisite proofs on runs of a system of interacting state machines to a set of definitions and checks on single steps of a small number of state machines corresponding to the intuitive notions of freedom from starvation and deadlock. We further refine the definitions to afford an efficient explicit-state checking procedure in certain finite state cases. We demonstrate the proof reduction on versions of the Bakery Algorithm.

1 Introduction

Much of hardware and software system design focuses on how to optimize the execution of tasks by dividing the tasks into smaller computations and then scheduling and distributing these computations on the available resources. The natural specification for these systems is an assurance that the systems eventually complete the supplied tasks with results consistent with an atomic (or as atomic as feasible) execution of the task. We refresh the notion of fair stuttering refinements [10] as a means of codifying these specifications – a fair stuttering refinement between two systems ensures that every infinite run of a lower-level system with fair selection and finite stuttering maps to a similarly restricted infinite run of a higher-level system. This notion of refinement can allow sequences of smaller steps in the implementation to be mapped to single steps in the specification while additionally requiring that every task makes progress to completion.

Many previous efforts [10] have attempted to improve the capability of theorem provers in reasoning about refinements for distributed and concurrent systems. Previous efforts in regards to the ACL2 theorem prover [4] focused on trying to reduce the proofs of stuttering refinements with additional structures added to define fair selection and ensuring progress. These efforts generally boiled down to showing that a specification could match the step of an implementation or the implementation stuttered and some `rank` function decreased. The primary difficulty in these proofs was defining and proving an inductive invariant (either through ACL2 or trying to prove the invariant through some form of state exploration). In addition, the inclusion of additional structures to track fairness and progress as well as the resulting definition of `rank` functions proved complex. Further, the additional structures at times obfuscated whether the specification was complete and accurate.

In this paper, we take a different tack. We assume certain characteristics of the system we are trying to verify and leverage these characteristics in reducing the proof obligations. In particular, we first assume that the systems we are trying to verify are asynchronous in terms of how tasks make progress to completion. Further, we require the system definition to split the normal next-state transition relation into a next-state relation which only takes forward steps and a blocking relation which defines precisely when a task is blocked from making progress. From these assumed characteristics, we define proof reductions

Warren Hunt Jr. and Anna Slobodova (Eds.):
ACL2 Theorem Prover and its Applications
EPTCS 249, 2017, pp. 78–94, doi:10.4204/EPTCS.249.6

which reduce the goal of proving fair stuttering refinement to proving properties of a few task steps in relation to each other. These proof reductions have been formally defined and mechanically proven in ACL2 and are included in the supporting materials for this paper. In the remainder of this paper, we will cover two stages of proof reductions, review the application of the reductions to a version of the Bakery Algorithm. We conclude the paper with further reductions targeting efficient automatic checks in the finite state case.

2 Preliminaries

Commonly, systems are defined by an initial state predicate: `(init x)` and a next-state relation: `(next x y)`. A run of the system is then simply a sequence of states where the first state satisfies `(init x)` and each pair of states in the sequence satisfies `(next x y)`. We extend this basic construction in a couple of ways.

First, our goal is to reason about fair executions of a system (either as an assumption of fair selection for which task will update next or as a guarantee that every task makes progress). Thus, we assume that there is some set of task identifiers recognized by a predicate `(id-p k)` and add a task id parameter to the next-state relation: `(next x y k)` where this now relates state `x` to state `y` for an update to the task with id `k`. We also assume only one task updates at each step of the system without any prescribed order of task updates – essentially, the system is asynchronous at the level of task updates.

Second, we will find it useful to require the definition of an additional relation `(blok x k)` which returns true when the task identified by `k` is currently blocked from making progress in state `x`. Further, with this required definition of `(blok x k)`, we will also require the theorem: `(not (next x x k))` be proven and use inequality of next-states as a marker that a task is making progress to completion.

A system is then defined by three functions: `(init x)`, `(next x y k)`, and `(blok x k)`. Our final goal is to prove that the fair runs of an implementation system map to fair runs of a specification system with an allotment for finite stuttering and some guarantee of progress. A run of a system is a function `(run i)` which takes a natural `i` and returns a state of the system. Runs will naturally need to satisfy some constraints as detailed in Figure 1. For a given system named `sys`, the macro `(def-inf-run sys)` assumes the definition of `(sys-init x)`, `(sys-next x y k)`, `(sys-blok x k)`, `(sys-pick i)`, `(sys-run i)` and generates the definitions and theorems defining the properties for the run as in Figure 1.

Of particular note, the function `(step x y k)` relates states `x` and `y` via `(next x y k)` only if `k` is not blocked in `x` and we are not stuttering (denoted by the input `k` being `nil`) – (as a note, the only requirement we place on `id-p` is that `(not (id-p nil))`). So, an infinite run is defined by two functions `(run i)` which defines the sequence of states and `(pick i)` which defines the sequence of task identifiers selected. We constrain `(pick i)` to only return an `id-p` or `nil`. We can now naturally define fair selection of `(pick i)` by positing the existence of a function `(fair k i)` which returns natural numbers and for each task id `k` will strictly decrease when `k` is not selected – see Figure 2. The macro `(def-fair-pick sys id-p)` assumes the definitions of `(sys-pick i)`, `(sys-fair k i)`, and `(id-p k)` and produces the theorems in Figure 2. We use the term *fair run* for an infinite run with a fair picker.

Fair selection of task identifiers ensures that each run only has finite stuttering and that each task gets a chance to make progress, but it does not guarantee that tasks actually make progress. We introduce the term *valid run* for a run which is not only fair but ensures progress for each task. In order to ensure progress, we define a function `(prog k i)` similar to `(fair k i)` but in addition to ensuring `pick`

```
(encapsulate
 ((run  (i) t)
  (pick (i) t))
 (local (defun run  (i) ....))
 (local (defun pick (i) ....))

 (defun step (x y k)
   (if (or (null k)    ;; finite stutter
           (blok x k)) ;; or k is blocked in x
       (equal x y)
     (next x y k)))

 (defthm run-init-thm (implies (zp i)   (init (run i))))
 (defthm run-step-thm (implies (posp i) (step (run (1- i)) (run i) (pick i))))
)
```

Figure 1: Definition of an infinite run in ACL2

```
(defthm fair-nat-thm (natp (fair k i)))

(defthm pick-fair-thm
  (implies (and (posp i)
                (id-p k)
                (not (equal (pick i) k)))
           (< (fair k i) (fair k (1- i)))))
```

Figure 2: *Fair Runs*: fair task selection during a run

```
(defthm prog-is-nat (natp (prog k i)))

(defthm run-prog-thm
  (implies (and (posp i)
                (id-p k)
                (or (not (equal (pick i) k))
                    (equal (run i) (run (1- i)))))
           (< (prog k i) (prog k (1- i)))))
```

Figure 3: *Valid Runs*: ensuring task progress during a run

eventually equals k, we also need to ensure that a state change actually occurs. The properties in Figure 3 ensure a *valid run* and the macro (def-valid-run sys id-p) produces these theorems for id-p, sys-run, sys-pick, and sys-prog. We note that a valid run is also a fair run and thus our notion of refinement is compositional – but it is better to prove that all fair runs of the implementation are valid runs and then restrict the refinement to valid runs mapping to valid runs and reduce the proof requirements accordingly at each step. This is straightforward from what we present in this paper but we do not focus on it in this paper.

3 Proof Reduction to Single System Steps

The principle objective of fair stuttering refinement is to prove that the fair runs of an implementation map to valid runs of a specification. The first set of proof reductions we present refresh similar attempts in past work [10, 8] in transferring these proof requirements on infinite runs to properties about single steps of two systems impl and spec. The difference between these past efforts and the work presented is that we directly specify properties related to guaranteeing progress for each task in the system and we leverage the definition of the blocking relation. In addition, while the proof reduction to single step presented in this section could be used as is, the design of the reduction is influenced by the needs of subsequent proof reductions over tasks presented in Section 4. The book "general-theory.lisp" in the supporting materials covers the work in this section.

The goal is to show that if one were to prove certain properties about steps of an implementation system impl and a specification system spec, then one could infer a *fair stuttering refinement* – every fair run of impl maps to a valid run of spec. We wish to prove this for any specification and implementation system, so specifically, for any impl and spec and any fair run impl-run of the implementation, if we have proven the required properties then we can map impl-run to a valid run spec-run of spec. An overview of the structure of the book "general-theory.lisp" is provided in Figure 4 and attempts to codify this goal. The definitions of the impl and spec systems and the fair run impl-run of impl are constrained within an encapsulate to only have the properties: (def-inf-run impl), (def-fair-pick impl id-p), (def-system-props impl id-p), (def-valid-system impl id-p), and (def-match-systems impl spec id-p). From this fair run impl-run and the properties proven on spec and impl, we can build a valid run spec-run. While it is not possible to make this a closed-form statement of correctness in ACL2, we believe the structure of the book is sufficient to establish the claim.

The function (spec-run i) in Figure 4 defines the spec state at each time to simply be (impl-map (impl-run i)) and the function (spec-pick i) is simply (impl-pick i) except that we introduce finite stutter (i.e. return nil) if the mapped state doesn't change. It is customary to define some notion of observation or labeling of states that must be preserved to ensure correlation of behavior between spec and impl – we assume human review has ensured that the mapping from impl states to spec states preserves any observations relevant to the specification. In this regard, it is relevant that the mapped run on the spec is relatively simple in definition as it avoids errors or oversights in specification due to an obfuscation of how the implementation and specification are correlated.

The properties we need to prove for impl and spec are defined by the macros def-system-props, def-valid-system, and def-match-systems. Along with the functions defining the impl and spec systems, additional definitions are required for each of these macros. We will shortly go into greater detail on the properties we will assume as constraints for these functions, but first, we refer to the listing provided in Figure 5.

```
(encapsulate
 (....) ;; constrained functions defining impl and spec.
 ....   ;; local def.s and prop.s to show constraints.

;; ASSUMPTIONS:
   ;; assume relevant properties of given systems impl and spec:
 (def-system-props  impl id-p)
 (def-valid-system  impl id-p)
 (def-match-systems impl spec id-p)
   ;; assume an infinite run of the impl system:
 (def-inf-run       impl)
 (def-fair-pick     impl id-p)
)

.... ;; def.s and theorems to establish results.

;; Define the corresponding (assumed to preserve "observations") spec run:
(defun spec-run (i) (impl-map (impl-run i)))

;; spec-pick will introduce stutter into spec-run when the mapped state doesn't change:
(defun spec-pick (i)
  (and (not (equal (impl-map (impl-run (1- i)))
                   (impl-map (impl-run i))))
       (impl-pick i)))

.... ;; additional def.s and theorems to establish results.

;; CONCLUSIONS:
   ;; and prove that the corresponding spec-run is indeed a valid run of spec:
(def-inf-run   spec)
(def-valid-run spec id-p)
```

Figure 4: Structure of the book ‘‘`general-theory.lisp`’’

- IMPL system definition:
  - `(impl-init x)` – initial predicate on states `x` for impl system
  - `(impl-next x y k)` – state `x` transitions to state `y` on selector `k`
  - `(impl-blok x k)` – state `x` blocked for transitions for selector `k`
- SPEC system definition:
  - `(spec-init x)` – initial predicate on states `x` for spec system
  - `(spec-next x y k)` – state `x` transitions to state `y` on selector `k`
  - `(spec-blok x k)` – state `x` blocked for transitions for selector `k`
- Definitions needed for `(def-system-props impl id-p)` macro:
  - `(impl-iinv x)` – inductive invariant for states in impl
- Definitions needed for `(def-match-systems impl spec id-p)` macro:
  - `(impl-map x)` – maps impl states to corresponding spec states
  - `(impl-rank k x)` – ordinal decreases until spec matches transition for `k`
- Definitions needed for `(def-valid-system impl id-p)` macro:
  - `(impl-noblk k x)` – is task id `k` invariantly unblocked in state `x`
  - `(impl-nstrv k x)` – ordinal decreases until `k` is in a `noblk` state
  - `(impl-starver k x)` – potential starver of `k` in `x` which is not blocked

Figure 5: Function Definitions for Single-Step System-Level Properties

The macro `(def-system-props impl id-p)` expands into simple theorems ensuring `(not (id-p nil))`, ensuring `(impl-next x x k)` is not valid, and ensuring the state predicate `(impl-iinv x)` is an inductive invariant for `impl` – namely that `(impl-iinv x)` holds in the initial state and persists across `(impl-next x y k)` transitions.

The `(def-match-systems impl spec id-p)` macro requires defining `(impl-map x)`, a mapping from impl states to spec states and a ranking function `(impl-rank k x)` which returns an ordinal for each task id `k`. The main properties generated by `def-match-systems` are the following:

```
(defthm map-matches-next
  (implies (and (impl-iinv x) (id-p k) (!= (impl-map x) (impl-map y))
                (impl-next x y k)
                (not (impl-blok x k)))
           (and (spec-next (impl-map x) (impl-map y) k)
                (not (spec-blok (impl-map x) k)))))

(defthm map-finite-stutter
  (implies (and (impl-iinv x) (id-p k) (= (impl-map x) (impl-map y))
                (impl-next x y k))
           (o<  (impl-rank k y) (impl-rank k x))))

(defthm map-rank-stable
  (implies (and (impl-iinv x) (id-p k) (id-p l) (!= k l)
                (impl-next x y l))
           (o<= (impl-rank k y) (impl-rank k x))))
```

The theorem `map-matches-next` ensures that on any step `(impl-next x y k)` for task `k` which is not blocked in `x` and where the mapped specification state changes (i.e. `(!= (impl-map x) (impl-map y))`) then the spec must be able to match the transition and the spec state cannot be blocked in the spec for task `k`. The theorem `map-finite-stutter` ensures that when the mapped implementation state does not change on an update for task `k` in impl, then the ordinal returned by `impl-rank` must strictly decrease and the theorem `map-rank-stable` ensures that this ordinal does not increase when task `k` is not selected. The clear intent of these properties is to ensure that as long as a task `k` is not indefinitely blocked when it is selected for update in impl, then eventually a matching spec transition must be generated. The question is then naturally how to ensure that a task is not indefinitely blocked. This concept of being indefinitely blocked is commonly called "starvation" in the literature and the `def-valid-system` macro will generate properties intended to ensure that no task is starved.

The `(def-valid-system impl id-p)` macro requires the definition of a predicate `(impl-noblk k x)` which is true when the task `k` can no longer be blocked in state `x` and a function `(impl-nstrv k x)` which nominally returns an ordinal that decreases until `(impl-noblk k x)` is true. Once a task `k` reaches an `impl-noblk` state, it can no longer be blocked until it transitions and thus the fair selection of `k` will ensure a transition of `k` occurs. Unfortunately, a task's progress to an `impl-noblk` state may be dependent on any number of other tasks or components in the impl state. At this general level of system definition, we only have system states `x` and task ids `k`, so we imagine that for any `k` and `x`, we could define a set of task ids called the *starve-set* which need to make progress before `k` can reach a `noblk` state. Updates to ids which are not in this starve-set should simply have no effect on this progress and so we will assume that `(impl-nstrv k x)` will strictly decrease on transitions for ids in the starve-set and remain unchanged otherwise. Unfortunately, it might be possible that all of the tasks in the *starve-set* are blocked and so we need the additional definition of an `(impl-starver k x)` which returns an id in this *starve-set* which is currently not blocked in state `x`. Additionally, we need to ensure that when an element outside of the starve-set is chosen, that the `(impl-starver k x)` remains unchanged. The

encoding of these properties as ACL2 theorems are generated from the `def-valid-system` macro and are listed here:

```
(defthm noblk-blk-thm
  (implies (and (iinv x) (id-p k)
                (noblk k x))
           (not (blok x k))))

(defthm noblk-inv-thm
  (implies (and (iinv x) (id-p k) (id-p l) (!= k l)
                (next x y l)
                (noblk k x))
           (noblk k y)))

(defthm starver-thm
  (implies (and (iinv x) (id-p k)
                (not (noblk k x)))
           (not (blok x (starver k x)))))

(defthm nstrv-decreases
  (implies (and (iinv x) (id-p k) (!= k (starver k x))
                (next x y (starver k x))
                (not (noblk k x)))
           (o< (nstrv k y) (nstrv k x))))

(defthm nstrv-holds
  (implies (and (iinv x) (id-p k) (id-p l) (!= k l)
                (next x y l)
                (not (noblk k x)))
           (o<= (nstrv k y) (nstrv k x))))

(defthm starver-persists
  (implies (and (iinv x) (id-p k) (id-p l) (!= k l) (!= l (starver k x))
                (next x y l)
                (not (noblk k x))
                (= (nstrv k y) (nstrv k x)))
           (= (starver k y) (starver k x))))
```

And with these properties assumed as constraints, we return to the goal of proving that the infinite run defined by `(spec-run i)` and `(spec-pick i)` from Figure 4 is indeed a valid run of `spec`. In order to do that we need to define a function `spec-prog` which satisfies the requirements set out in Figure 3. First, it is useful to define an `(impl-prog k i)` and show that the `impl-run` is indeed a valid run.

The definition of `(impl-prog k i)` is in Figure 6 and essentially looks forward into `impl-run` until we reach an `i` where `k` is picked and the state changes. The key point is obviously the question of what is the measure for demonstrating that this function terminates and this follows from our earlier discussion about the `(impl-noblk k x)`, `(impl-nstrv k x)`, and `(impl-starver k x)` functions. If we have `(impl-noblk k ..)` at the current state, then the task with id `k` cannot be blocked and we can simply countdown the `(impl-fair k i)` measure until task `k` is selected – the state will change at that time since `k` will still be unblocked and `impl-next` must change the state. If `(impl-noblk k ..)` does not currently hold then we know there is a task id `(impl-starver k ..)` which cannot be blocked in the current state and either `(impl-nstrv k ..)` strictly decreases or `(impl-starver k ..)` will not change. Thus, at each step, either the `impl-nstrv` measure strictly decreases or the fair measure for `impl-starver` counts down and will eventually expire and `impl-nstrv` will strictly decrease.

This `(impl-prog k i)` thus ensures that task `k` is picked and changes state in `(impl-run i)` but we now must guarantee that the mapped state changes in `spec`. In the case that the mapped state doesn't change, we know that the `(impl-rank k ..)` must decrease and that the impl-rank remains unchanged

```
(defun ord-nat-pair (o n)
  ;; simple function which returns lex. product of an o-p o and natp n:
  (make-ord (if (atom o) (1+ o) o) 1 n))

;; First prove that the implementation run is a valid run...
(defun impl-prog (k i)
  (declare (xargs :measure
                  (if (impl-noblk k (impl-run i))
                      (impl-fair k i)
                    (ord-nat-pair (impl-nstrv k (impl-run i))
                                  (impl-fair (impl-starver k (impl-run i)) i)))))
  (cond
   ((or (not (and (natp i) (id-p k)))              ;; ill-formed inputs.. or
        (and (= (impl-pick (1+ i)) k)              ;;    impl-pick matches k
             (!= (impl-run (1+ i)) (impl-run i)))) ;;  ..and k makes progress
    0)
   (t (1+ (impl-prog k (1+ i))))))

;; ...And use that to show that the mapped spec run is also valid
(defun spec-prog (k i)
  (declare (xargs :measure
                  (ord-nat-pair (impl-rank k (impl-run i))
                                (impl-prog k i))))
  (cond
   ((or (not (and (natp i) (id-p k)))              ;; ill-formed inputs.. or
        (and (= (spec-pick (1+ i)) k)              ;;    spec-pick matches k
             (!= (spec-run (1+ i)) (spec-run i)))) ;;  ..and k makes progress
    0)
   (t (1+ (spec-prog k (1+ i))))))
```

Figure 6: Defined Measure Functions on Infinite Runs

when other ids are selected. This is the basis for the definition `(spec-prog k i)` in Figure 6.

4 Proof Reduction to a Small Bounded Number of Tasks

In the previous section, we presented a proof reduction of the requirements for fair stuttering refinement from reasoning about infinite runs of systems to reasoning about single steps of systems. We did not make any assumption about the state structure of the systems other than that updates occurred asynchronously at some prescribed task level. In this section, we will assume a structure on the states of a system and show how to reduce the requisite properties from across the large state structure to the properties on components of the state. Throughout this section and the next, we will use the set `(s k v r)` and get `(g k r)` operations from the records book [5]. In particular, `(g k r)` takes a record `r` and returns either the value previously set for key `k` in record `r` or `nil` as default.

The book "trans-theory.lisp" in the supporting materials for this paper includes the definitions and proofs relating to this section. The structure of this book is similar to that shown for "general-theory.lisp" in Figure 4 in that there is an encapsulation which entails the system definitions and properties we want to assume and then outside of the encapsulation, we prove the derived results. For the previous section, in "general-theory.lisp", we proved the property in Figure 8 (in an abuse of notation pretending ACL2 were higher-order for a moment), For this section, our goal is to define systems at a task level and derive the system-level results. In the same higher-level-abuse format as above, we have the property from "trans-theory.lisp" also in Figure 8.

We take the state of the system to be a record associating keys to task states.. what we call *t-states*. The task id selected on input is now simply one of these keys and the update of the state will only update

- TR-IMPL system definition:
  - `(tr-impl-t-init a k)` – initial state predicate for t-state `a` and key `k`
  - `(tr-impl-t-next a b x)` – t-state `a` transitions to t-state `b` in state `x`
  - `(tr-impl-t-blok a b)` – t-state `a` is blocked from stepping by t-state `b`
- TR-SPEC system definition:
  - `(tr-spec-t-init a k)` – initial state predicate for t-state `a` and key `k`
  - `(tr-spec-t-next a b x)` – t-state `a` transitions to t-state `b` in state `x`
  - `(tr-spec-t-blok a b)` – t-state `a` is blocked from stepping by t-state `b`
- Definitions needed for `(def-tr-system-props tr-impl)` macro:
  - `(tr-impl-iinv x)` – inductive invariant as previously.. no change at task-level
- Definitions needed for `(def-match-tr-systems tr-impl tr-spec)` macro:
  - `(tr-impl-t-map a)` – maps tr-impl t-states to corresponding tr-spec t-states
  - `(tr-impl-t-rank a)` – ordinal decreases until mapped t-state must change
- Definitions needed for `(def-valid-tr-system tr-impl)` macro:
  - `(tr-impl-t-noblk a b)` – is t-state `a` invariantly not-blocked by t-state `b`
  - `(tr-impl-t-nstrv a b)` – positive natural which strictly decreases until `(t-noblk a b)`
  - `(tr-impl-t-nlock k x)` – ordinal strictly decreases on from `k` to blocker of `k` in `x`

Figure 7: Function Definitions for Single-Step Task-Level Properties

```
"general-theory.lisp":
  (implies (and (def-system-props  impl id-p)
                (def-valid-system  impl id-p)
                (def-match-systems impl spec id-p))
           (implies (and (def-inf-run   impl)
                         (def-fair-pick impl id-p))
                    (and (def-inf-run   spec)
                         (def-valid-run spec id-p))))

"trans-theory.lisp":
  (implies (and (def-tr-system-props  tr-impl)
                (def-valid-tr-system  tr-impl)
                (def-match-tr-systems tr-impl tr-spec))
           (and (def-system-props  tr-impl key-p)
                (def-valid-system  tr-impl key-p)
                (def-match-systems tr-impl tr-spec key-p)))
```

Figure 8: High-Level properties in for theory files definitions

the corresponding entry of the record. We presume and constrain a fixed finite set of keys – `(keys)` – of arbitrary size and composition and membership in this set will define the `id-p` test for task id selection. The state of the system is then a record mapping members of this finite set `(keys)` to t-states and the system will be defined on the task level. We define task-based systems by assuming the pertinent definitions on task states in the system and derive the system-level definitions across the state. We name these systems derived from the task-level definitions as `tr-impl` and `tr-spec`. In Figure 5 from the previous section, we listed the function definitions required for the single-step system-level properties – we do the same for the single-step task-level properties in Figure 7.

Many of the system-level derived functions follow simply from the task-level. The system-level `(tr-impl-init x)` predicate checks that `(tr-impl-t-init (g k x) k)` holds for all keys `k`. The system-level `(tr-impl-next x y k)` only updates `(g k x)` as `(tr-impl-t-next (g k x) (g k y) x)` and leaves all other keys untouched in `x`. The system-level block function `(tr-impl-blok x k)` checks if there is any key `l` such that `(tr-impl-t-blok (g k x) (g l x))`. The system-level mapping function simply goes through all keys and calls `tr-impl-t-map` for the corresponding t-state and the system level rank just calls `(tr-impl-t-rank (g k x))` directly. The inductive invariant does not change; there is just one inductive invariant defined on the entire record defining the system state. Additionally, the system-level proofs for `(def-system-props tr-impl key-p)` and `(def-match-systems tr-impl tr-spec key-p)` are straightforward and follow from these system-level definitions and properties of task-level definitions.

The functions and properties for proving progress and valid `impl` runs are more involved. For the sake of brevity and readability, we will drop the `tr-impl-` prefix from the system-level and task-level defintions for the remainder of this section. In addition to ensuring that `t-nlock` returns an ordinal and `t-nstrv` returns a positive natural number[1], the macro `(def-valid-tr-system tr-impl)` introduces the following properties:

```
(defthm t-noblk-blk-thm
  (implies (and (iinv x) (key-p k) (key-p l)
                (t-noblk (g k x) (g l x)))
           (not (t-blok (g k x) (g l x)))))

(defthm t-noblk-inv-thm
  (implies (and (iinv x) (key-p k) (key-p l)
                (t-noblk (g k x) (g l x))
                (t-next (g l x) c x))
           (t-noblk (g k x) c)))

(defthm t-nlock-decreases
  (implies (and (iinv x) (key-p k) (key-p l)
                (t-blok (g k x) (g l x)))
           (o< (t-nlock l x)
               (t-nlock k x))))

(defthm t-nstrv-decreases
  (implies (and (iinv x) (key-p k) (key-p l)
                (not (t-noblk (g k x) (g l x)))
                (not (t-noblk (g k x) c))
                (t-next (g l x) c x))
           (< (t-nstrv (g k x) c)
              (t-nstrv (g k x) (g l x)))))
```

[1] In the supporting materials for this paper, `t-nstrv` is generalized to be a list of natural numbers which is then summed and combined into a list of lists of naturals, but for the sake of clarity and brevity in this paper, we keep a simpler definition for `t-nstrv`. We could not use a generic ACL2 ordinal for `t-nstrv` since we needed to form lexicographic products of sums of these ordinals and that is not possible for arbitrary ordinals in ACL2.

The system-level (noblk k x) definition simply checks that (t-noblk (g k x) (g l x)) holds for every key l and as such, the task-level t-noblk-blk-thm and t-noblk-inv-thm are task-level projections of their system-level counterparts and the system-level properties follow fairly easily. The more interesting case comes up in defining the system-level (nstrv k x) and (starver k x). For the task-level, the property t-nlock-decreases ensures that we don't have any "deadlocks" or simply that for any set of keys, there is always some key in that set which is not blocked in x by some other key in that set. The combination of t-nstrv-decreases and the properties of t-noblk ensure that no task can be starved by another task.

The intuition behind defining the system-level (nstrv k x) begins by recognizing that if (not (noblk k x)) then there is some set of keys l such that (not (t-noblk (g k x) (g l x))). We will call this set of keys the *may-block set*. But since t-noblk persists once we reach it, then we could sum up the (t-nstrv (g k x) (g l x)) for this may-block set and the resulting ordinal would decrease until we reached a state where k was t-noblk for all l and thus noblk. Assume for the moment that k were not blocked (i.e. we could set (starver k x) to be k), then consider an update for some key l. If that key were in the may-block set of k then the ordinal would decrease. If l is not in the may-block set of k then (t-noblk (g k x) (g l x)) and the transition of l cannot change the blocked status of k and it cannot change the may-block set for k and so progress is made. Unfortunately there is no guarantee that k is not blocked and thus we cannot pick a suitable starver which ensures progress when selected.

But from the property t-nlock-decreases, starting with k in x, we can find a key which is not blocked by checking if the key is blocked and recurring on the first blocking key we find if we are blocked. This is the definition of the function (starver k x) and is included here:

```
(defun starver (k x)
  (declare (xargs :measure (t-nlock (g k x))))
  (if (and (iinv x) (key-p k) (blok x k))
      (starver (pikblk k x) x)
    k))
```

The function (pikblk k x) returns the first key we find such that (t-blok (g k x) (g (pikblk k x))). So, from k, we can find a key which is unblocked, but the question is then how to build a measure from the starve-set including k and (starver k x). The answer is to build a natural list where each element is the sum of t-nstrv for the may-block set (as we described before) in each step along the path from k to (starver k x) and define our ordinal as the lexicographic product of the naturals in this list. The first observation is that at the end of this list we will have the summation of t-nstrvs for the may-block set of (starver k x) and since (starver k x) is not blocked, it will make progress as we discussed before. The other key observation is that at each step, the (pikblk k x) key will be in the may-block set of k and thus even though a transition of (pikblk k x) may modify its may-block set and potentially increase the measure from that point, the measure for the may-block set of k will decrease and the ordinal over all will decrease. This list of naturals is defined by the function (nstrvs* k x) as follows where the function (scar s) and (scdr s) return the first element and remainder of a set respectively and (card s) returns the cardinality of the set.

```
(defun sum-nsts* (k x s)
  (declare (xargs :measure (card s)))
  (if (null s) 1
    (+ (if (t-noblk (g k x) (g (scar s) x)) 0
         (t-nstrv (g k x) (g (scar s) x)))
       (sum-nsts* k x (scdr s)))))

(defun sum-nsts (k x) (sum-nsts* k x (keys)))

(defun nstrvs* (k x)
  (declare (xargs :measure (t-nlock (g k x))))
  (if (and (iinv x) (key-p k) (blok x k))
      (cons (sum-nsts k x) (nstrvs* (pikblk k x) x))
    (list (sum-nsts k x))))

(defun nats->o (n l)
  (cond ((zp n) 0)
        ((atom l) (make-ord n 1            (nats->o (1- n) ())))
        (t        (make-ord n (1+ (car l)) (nats->o (1- n) (cdr l))))))

(defun tr-impl-nstrv (k x)
  (nats->o (card (keys)) (nstrvs* k x)))
```

As we mentioned, the function `nstrvs*` returns a natural list and we build a suitable ordinal from this list using the function `nats-o`. But because the length of the path to `(starver k x)` from `k` could change and thus the length of the nstrvs* list could change, we need to make the defined ordinal "first-aligned" – where the first element in the list is mapped to a coefficient of the same exponent no matter the length of the rest of the list. We use `(card keys)` as the starting exponent and prove separately that the length of the list returned by `nstrvs*` can never exceed `(card keys)`.

This construction also shows one of the reasons we assume an arbitrary fixed finite set of `(keys)` (in order to put a bound on `(len (nstrv* k x))`), but this restriction makes sense for other reasons as well. If the set of keys were not finite, then we would need some additional requirement to ensure that a task were not persistently blocked by an infinite sequence of newly instantiated tasks. Other options exist to avoid this (such as requiring that all new tasks cannot block existing tasks) but these alternatives end up imposing constraints we believe are too restrictive.

5 Example – A Bakery Algorithm

We use the Bakery algorithm as an example application of the proof reductions we present in this paper. The Bakery algorithm was developed by Lamport [7] as a solution to mutual exclusion with the additional assurance that every task would eventually gain access to its exclusive section. The Bakery algorithm has also been a focus of previous ACL2 proof efforts [9].

The essential idea of the algorithm is that each task first goes through a phase where it chooses a number (much like choosing a number in a bakery) and then later compares the number against the numbers chosen by the other tasks to determine who should have access to the exclusive section. The version of the Bakery algorithm we will use is defined in Figure 9 (the `(upd r .. updates ..)` simply expands into a nest of record sets).

Each task will start in program location 0 and start its `:choosing` phase. During the `:choosing` phase, the task will grab the current shared max (via the function `(curr-sh-max x)`) and then set its

```
(defun bake-impl-t-init (a k)
  (= a (upd nil :loc 0 :key k :pos 1 :old-pos 0 :temp 0 :sh-max 1)))

(defun bake-impl-t-next (a b x)
  (case (g :loc a)
        (0 (= b (upd a :loc 1 :choosing t)))
        (1 (= b (upd a :loc 2 :temp (curr-sh-max x))))
        (2 (= b (upd a :loc 3 :pos (1+ (g :temp a))
                              :old-pos (g :pos a)
                              :pos-valid t)))
        (3 (= b (upd a :loc 4 :sh-max (if (> (curr-sh-max x) (g :temp a))
                                          (curr-sh-max x)
                                        (g :pos a)))))
        (4 (= b (upd a :loc 5 :choosing nil)))
        (5 (= b (upd a :loc 6))) ;; we are potentially blocked here
        (6 (= b (upd a :loc 7))) ;; we are potentially blocked here
        (t (= b (upd a :loc 0 :pos-valid nil)))))

(defun bake-impl-t-blok (a b)
  (or (and (= (g :loc a) 5)
           (g :choosing b))
      (and (= (g :loc a) 6)
           (and (g :pos-valid b)
                (lex< (g :pos b) (ndx (g :key b))
                      (g :pos a) (ndx (g :key a)))))))
```

Figure 9: Bakery Implementation System Definition

own position `:pos` to be 1 more than the shared max. In program `:loc` 3, a compare-and-swap is implemented and the shared-max is potentially updated. The task then ends its `:choosing` phase.

After the `:choosing` phase, the task will enter program locations 5 and 6. In these locations, the `t-blok` predicate ensures that the task wait until other tasks are not `:choosing` and then wait until it has the least position (where potential ties are broken by comparing the `ndx` of the `:key` in the set `(keys)`.

In order to prove `(def-valid-tr-system bake-impl)`, we need to define the `t-nlock`, `t-noblk`, and `t-nstrv` functions. The definition of `(t-nlock x k)` needs to return an ordinal that is strictly decreasing from the blocked task to the blocking task. From the `bake-impl-t-blok` relation, we note that `:choosing` states cannot be blocked and that `lex<` is already well-founded, so we can devise a suitable `bake-impl-t-nlock`:

```
(defun bake-impl-t-nlock (k x)
  (let ((a (g k x)))
    (make-ord 2 (if (g :choosing a) 1 2)
    (make-ord 1 (1+ (nfix (g :pos a)))
                (ndx (g :key a))))))
```

For the `t-noblk` and `t-nstrv` definitions, we need to analyze where one task can no longer block another task. The simple answer is that `(t-noblk a b)` is reached once task `b` has chosen a `:pos` greater than the one in `a`, but we also have to make sure that task `b` is not choosing either. In addition, we note that if `a` cannot currently be blocked by any task, then we can set `t-noblk` and task `a` cannot be blocked if it is not in program locations 5 or 6. With that, we define `bake-impl-t-noblk`:

```
(defun bake-impl-t-noblk (a b)
  (or (and (!= (g :loc a) 5)
           (!= (g :loc a) 6))
      (and (not (g :choosing b))
           (> (g :pos b) (g :pos a)))))
```

Finally, we need to define t-nstrv which counts down until we reach the t-noblk state. The simple answer would be to count from the exit of :choosing phase until the next exit from the :choosing phase. Thus, we would return 8 if (g :loc b) was 5 and then proceed down to 6 for 7, then 5 for 0 (wrapping back), then down to 1 for 4 (end of next :choosing). This almost works.. except that it is possible for b to be in :loc 2, 3, or 4 with a :pos lower than a but a has proceeded further. Thus, we need to add a few steps for the case of being in 2,3,4 with a potentially lower :pos but when we come back around for the next :choosing, we will reach noblk:

```
(defun bake-impl-t-nstrv (a b)
  (pos-fix
   (cond ((or (and (= (g :loc b) 2)
                   (< (g :temp b) (g :pos a)))
              (and (> (g :loc b) 2)
                   (<= (g :pos b) (g :pos a))))
          (+ 8 (- 8 (g :loc b))))
         ((>= (g :loc b) 5)
          (+ 5 (- 8 (g :loc b))))
         (t
          (+ 0 (- 5 (g :loc b)))))))
```

With these definitions and a suitable invariant bake-impl-iinv, we can prove the theorems for (def-valid-tr-system bake-impl) – each of which just blasts into a big case split which pushes through. For the specification of the bakery algorithm, we have a simple system bake-spec defined in Figure 10. Each task in this system goes through the following steps: first, load up a new provisional :pos in the :load variable, then proceed to set the :pos variable and begin to arbitrate in the 'interested state. Tasks are blocked if some other task is in the 'go state or is in the 'interested state and has a lower :pos. The definitions and proof of (def-match-tr-systems bake-impl bake-spec) are fairly straightforward and included in Figure 10. We note that it is feasible (although not required) to define the supporting functions and prove (def-valid-tr-system bake-spec) – this proves that all fair runs of bake-spec are valid while the earlier proofs only ensured that the runs mapped from bake-impl runs were valid.

In previous work [10], a similar proof effort was conducted in proving a fair stuttering refinement for the definition of the Bakery Algorithm. In that effort, the proof was complicated by the need to add additional structures to track fair scheduling and to ensure correlation to a specification which had additional structures to ensure progress for each task. These complications were avoided in the proof here and as such, much less definition and details were required. The reduced proof we present here is primarily the definition and proof of a sufficient inductive invariant but much additional definition and proof was required in the earlier work [10].

6 Further Reductions and Considerations

We conclude this paper with a discussion of further reductions and considerations for search procedures. We first acknowledge that some of the task-based definitions may seem overly restrictive. For example, the (blok a b) relation being defined simply on task states. In essence, this restricts us from supporting systems where a task may be blocked when only some combination of tasks exist. It is possible to extend the notion of blocking to be more general but it comes at the cost of the complexity of other definitions and checks and we have generally found that by adding auxiliary variables to the task state, we can fit any appropriate system under these restrictions.

```
(defun bake-spec-t-init (a k)
  (declare (ignore k))
  (and (= (g :loc a) 'idle) (= (g :pos a) 0) (= (g :load a) 0)))

(defun bake-spec-t-next (a b x)
  (case (g :loc a)
    (idle       (and (= (g :loc b)  'loaded)
                     (= (g :pos b)  (g :pos a))
                     (natp (g :load b))
                     (>  (g :load b) (max-pos x))
                     (>= (g :load b) (max-load x))))
    (loaded     (= b (upd a :loc 'interested
                            :pos (g :load a))))
    (interested (= b (upd a :loc 'go)))
    (go         (= b (upd a :loc 'idle)))))

(defun bake-spec-t-blok (a b)
  (and (= (g :loc a) 'interested)
       (or (= (g :loc b) 'go)
           (and (= (g :loc b) 'interested)
                (< (g :pos b) (g :pos a))))))

(defun bake-impl-t-map (a)
  (upd nil
       :loc  (case (g :loc a)
                   ((0 1) 'idle)
                   ((2 3) 'loaded)
                   ((4 5 6) 'interested)
                   (t 'go))
       :pos  (case (g :loc a)
                   (3 (g :old-pos a))
                   (t (g :pos a)))
       :load (case (g :loc a)
                   (2 (1+ (g :temp a)))
                   (t (g :pos a)))))

(defun bake-impl-t-rank (a)
  (case (g :loc a)
        (0 1)       (1 0)
        (2 1)       (3 0)
        (4 2) (5 1) (6 0)
        (t 0)))
```

Figure 10: Bakery Specification System and Definitions for Proving Matching from Impl

This paper focused on mechanized proof reductions for general system definitions, but the work also supports improvements in more efficient automatic verification (in particular when the underlying task state space is finite). For example, take a somewhat draconian restriction that `(t-next a b x)` can be defined as `(t-next a b)` and similarly, the initial state predicate ignored the input `k` – a few things develop in this case. First, we note (somewhat trivially) that for every reachable system state composed of (say) n task states, that every "substate" of $n-1$ task states can also be reached. Additionally, if the task state space were finite, then we could compute all of the potential cycles in the blocking relation and for each cycle of size n, we could determine if it was reachable by searching through the system states with only n keys. A similar check could be implemented for the other properties with no more than 2 keys needed.

Of additional interest in this case, is that reachable states of these systems have a particular characterization. Consider any run of a system.. any steps in the run can be permuted as long as the permutation does not change the blocking relationship between the tasks involved. This means that for every reachable state, one can define a set of canonical runs which involves only stepping tasks until the blocking relationship is changed with respect to another task and then switching to the blocking task or stepping back and switching to the blockee task. This property limits the structure of potential invariants and suggests procedures for proving invariants over pairs of states. The inductive invariant `iinv` over the system state can be defined by invariant definitions on single task states, pairs of states, triples, etc. and in most cases (potentially with additional auxiliary variables), sufficiently defined on single t-states and pairs of t-states. In this case, the requisite properties of the defined `t-nlock`, `t-nstrv`, `t-noblk`, `t-map` and `t-rank` definitions could be proven via GL on the specified finite t-state domain using a SAT solver with a sufficient conditions on the t-states assumed. An inductive invariant (defined on single t-states and pairs of t-states) could be defined that proved each of these sufficient condition assumptions as invariant of the system. A model checker could be used to reduce the definitional requirements further by checking invariants (not requiring inductive invariants) and by checking for bad cycles to show that one could infer the existence of suitable `t-nlock`, `t-nstrv`, and `t-rank`. The model checking problems could be limited to a small number of tasks and possibly only single task stepping depending on the conditions of the defintion. The work presented in this paper is a step into many potential future directions.

References

[1] Susanne Graf & Hassen Saïdi (1997): *Construction of Abstract State Graphs with PVS*. In: *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, pp. 72–83, doi:10.1007/3-540-63166-6_10.

[2] 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*. *J. Funct. Program.* 18(1), pp. 15–46, doi:10.1017/S0956796807006338.

[3] Mitesh Jain & Panagiotis Manolios (2015): *Proving Skipping Refinement with ACL2s*. In: *Proceedings Thirteenth International Workshop on the ACL2 Theorem Prover and Its Applications, Austin, Texas, USA, 1-2 October 2015.*, pp. 111–127, doi:10.4204/EPTCS.192.9.

[4] Matt Kaufmann, Panagiotis Manolios & J Strother Moore (2000): *Computer-Aided Reasoning: An Approach*. Kluwer Academic, doi:10.1007/978-1-4757-3188-0.

[5] Matt Kaufmann & Rob Sumners (2002): *Efficient rewriting of data structures in ACL2*. In Kaufmann M. Moore J.S. Borrione, D., editor: *Proceedings of 3rd International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2002)*.

[6] Marcel Kyas & Jozef Hooman (2006): *Compositional Verification of Timed Components using PVS*. In: *Software Engineering 2006, Fachtagung des GI-Fachbereichs Softwaretechnik, 28.-31.3.2006 in Leipzig*, pp. 143–154.

[7] Leslie Lamport (1974): *A New Solution of Dijkstra's Concurrent Programming Problem*. *Commun. ACM* 17(8), pp. 453–455, doi:10.1145/361082.361093.

[8] Panagiotis Manolios, Kedar S. Namjoshi & Robert Summers (1999): *Linking Theorem Proving and Model-Checking with Well-Founded Bisimulation*. In: *Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings*, pp. 369–379, doi:10.1007/3-540-48683-6_32.

[9] Sandip Ray & Rob Sumners (2011): *A theorem proving approach for verification of reactive concurrent programs*. In Chaudhury S. Farzan A. Gopalakrishnen G. Seigel S. Burckhardt, S., editor: *4th International Workshop on Exploiting Concurrency Efficiently and Correctly (EC2 2011)*.

[10] Sandip Ray & Rob Sumners (2013): *Specification and Verification of Concurrent Programs Through Refinements*. *J. Autom. Reasoning* 51(3), pp. 241–280, doi:10.1007/s10817-012-9258-1.

Term-Level Reasoning in Support of Bit-blasting

Sol Swords
Centaur Techology, Inc.
Austin, TX, USA
sswords@centtech.com

GL is a verified tool for proving ACL2 theorems using Boolean methods such as BDD reasoning and satisfiability checking. In its typical operation, GL recursively traverses a term, computing a symbolic object representing the value of each subterm. In older versions of GL, such a symbolic object could use Boolean functions to compactly represent many possible values for integer and Boolean subfields, but otherwise needed to reflect the concrete structure of all possible values that its term might take. When a term has many possible values that can't share such a representation, this can easily cause blowups because GL must then case-split. To address this problem, we have added several features to GL that allow it to reason about term-like symbolic objects using various forms of rewriting. These features allow GL to be programmed with rules much like the ACL2 rewriter, so that users may choose a better normal form for terms for which the default, value-like representation would otherwise cause case explosions. In this paper we describe these new features; as a motivating example, we show how to program the rewriter to reason effectively about the theory of records.

1 Introduction

GL is an ACL2 framework for proving theorems using bit-level methods, namely BDD reasoning or SAT solving [6, 5]. It works by computing bit-level symbolic representations of terms, which we also describe as symbolically executing them. To prove a theorem, it computes such a representation of the theorem's conclusion and thus reduces it to a satisfiability check: if the negation of the Boolean formula representing the conclusion is unsatisfiable, then the theorem is proved.

This paper introduces term-level reasoning capabilities that are now built into GL. These capabilities allow GL to prove theorems at a higher level of abstraction, without computing a concrete representation for the value of each subterm encountered. It also allows theorems to be proved about unconstrained variables by generating bit-level representations for accessors applied to those variables. We have verified the soundness of all of these extensions in order to admit GL as a verified clause processor.

1.1 Extensions to "Traditional" GL

GL has always contained a symbolic object representation for function calls and unconstrained variables, but previously the use of such objects would almost guarantee failure. GL's previous use was *value-oriented*, as indicated by the use of the term "symbolic execution": the GL system would crawl over a term just as a simple applicative Lisp interpreter would, producing a (symbolic) value for each subterm. Typical symbolic values expected to appear during a successful symbolic execution could include:

- symbolic Booleans – Boolean functions in a BDD or AIG representation, depending on the mode of operation
- symbolic integers – represented as a list of Boolean functions, one for each two's-complement bit of the integer

Warren Hunt Jr. and Anna Slobodova (Eds.):
ACL2 Theorem Prover and its Applications
EPTCS 249, 2017, pp. 95–111, doi:10.4204/EPTCS.249.7

- concrete values – represented by themselves (except for objects containing certain tagging symbols used to delineate kinds of symbolic objects, which could then be tagged to "escape" them)
- conses of symbolic values.

In certain cases the symbolic values might also be expected to contain:

- if-then-else constructs, so that a symbolic value could take various different concrete values or different types of symbolic value depending on symbolic conditions.

But two other kinds of symbolic objects existed in the representation primarily for use in bad cases:

- function call – representing a function applied to a list of symbolic values, primarily used to indicate that GL was unable to compute a useful representation for some subterm
- variable – representing a free variable of unknown type, used when a theorem variable lacked a specific symbolic representation provided by the user.

The new GL features described in this paper allow it to operate in a *term-oriented* manner, where the above function and variable representations can also be manipulated productively. We describe the new features supporting this in Section 2 and discuss some implementation details that may be relevant to users in Section 3. We show a complete set of rules for term-level manipulation of records in Section 4, then describe how this new feature set has been applied to a class of industrial proofs in Section 5. We discuss some issues involved in the proof of soundness of these new features in Section 6. Some possible future extensions are discussed in Section 7.

2 Term-level Features of GL

The term-level capabilities of GL consist of the following new features. While complete documentation of these features is beyond the scope of this paper (see the ACL2 community books documentation [1], including topic gl::term-level-reasoning), we motivate and briefly describe each of them in this section:

- Conditional rewriting with user-defined rules integrated with the symbolic interpreter (Section 2.1)
- Generation and tracking of fresh Boolean variables representing truth values of term-level objects (Section 2.2)
- Rules to generate counterexamples from Boolean valuations for terms (Section 2.3)
- Rules to add constraints among the generated Boolean variables (Section 2.4)
- Rules that merge `if` branches containing different term-level objects that could be represented in a common way (Section 2.5)
- Support for binding theorem variables to shape specifiers representing function calls (Section 2.6).

2.1 Abstracting Implementation Details with Rewriting

Suppose we are using the popular records library found in the community books file "misc/records.lisp" [4], and we have the term `(s :a b nil)`. Supposing *b* is a 4-bit integer, and we have its symbolic representation as a vector of 4 Boolean functions, we can represent this call as `'((:a . b))`. But suppose instead that *b* is a Boolean. The default value of any key in a record is `nil`, so the `s` function does not add pairs to record structures when binding them to `nil`. So the best we can do to represent this value is a term-like representation,

```
(if b '((:a . t)) 'nil)
```

This effectively introduces a case-split: every further operation on this record must consider separately the case where *b* is `t` versus `nil`. We can easily get an exponential blowup in our representation by setting several keys to symbolic Boolean values. This is especially annoying since the elision of pairs that bind keys to `NIL` is just an implementation detail of the library, not particularly relevant to the meaning of what is happening.

A better way to deal with this is to avoid representing the record as a cons structure. Instead, GL can treat the record library's `s` and `g` as uninterpreted, and use rewrite rules to resolve them. The following forms declare these functions uninterpreted:

```
(gl::gl-set-uninterpreted g)
(gl::gl-set-uninterpreted s)
```

When the symbolic interpreter reaches a call of `g` or `s`, it will compute symbolic representations for the arguments to the call. Normally, it would then symbolically interpret the definition of the function. These declarations say to instead simply return an object representing the call of `g` or `s` applied to those symbolic arguments. These function call objects can then be operated on by rewrite rules.

GL's rewrite rules may be declared using `gl::def-gl-rewrite`; this stores the theorem as an ACL2 rewrite rule but disables it and adds it to a table that determines which rules GL will try. For records, some proofs can be completed with only the case-splitting version of the simple get-of-set rule:

```
(gl::def-gl-rewrite g-of-s-for-gl
 (equal (g k1 (s k2 v r))
        (if (equal k1 k2)
            v
          (g k1 r))))
```

In general, a more comprehensive set of rules is required; we discuss these in Section 4. With this rule in place, if a `g` call is encountered and the symbolic representation of its second argument is a call of `s`, then this rule applies, and instead of producing a `g` call object, the symbolic interpreter is instead applied to the right-hand side of this rule with the appropriate substitution.

GL's rewriter supports a subset of the ACL2 rewriter's features. It can use rules that rewrite on `equal` or `iff` equivalences. It supports conditional rewriting by backchaining to relieve hypotheses. It supports `syntaxp` hypotheses, but these aren't compatible with ACL2's syntaxp hypotheses, because ACL2's use term syntax whereas GL's use symbolic object syntax. Some further improvements may be considered in future work, which we discuss in Section 7.2.

2.2 Boolean Representations for Unconstrained Variables and Accessors

Traditionally, most theorems proved using GL have had hypotheses constraining each variable to a type that could be represented using a bounded number of bits, such as `(unsigned-byte-p 8 x)`. But in some cases this shouldn't be necessary because only certain bits are accessed from the variable regardless of its size or type. Consider:

```
(equal (lognot (lognot (loghead 5 x))) (loghead 5 x))
```

This is true for any value of `x`, even non-integers; but previously, GL would not be able to prove this without constraining `x` to be a member of a bounded type. However, GL's new term-level capabilities include the ability to generate fresh Boolean variables to represent certain terms. Specifically, GL generates a fresh Boolean variable for any `if` test that cannot be otherwise resolved to a symbolic Boolean

value. Therefore, a rewrite rule such as the following may be used to introduce new Boolean variables for `loghead` calls such as those above. (Note that the syntaxp test in this rule applies to the GL symbolic object representation, not the ACL2 term representation!)

```
(gl::def-gl-rewrite expand-loghead-bits
  (implies (syntaxp (integerp n))
           (equal (loghead n x)
                  (if (zp n)
                      0
                    (logcons (if (logbitp 0 x) 1 0)
                             (loghead (1- n) (ash x -1)))))))
```

This rewrite rule unwinds the `loghead` call of the theorem above. Each `logbitp` call first simply produces a term-level representation, but then, since the term is used as an `if` test, a new Boolean variable is generated to represent it. In particular, Boolean variables are introduced for the following terms:

```
(logbitp 0 x)
(logbitp 0 (ash x -1))
(logbitp 0 (ash (ash x -1) -1)) ...
```

We can tweak this representation via the following rewrite rule to a more uniform `(logbitp` *n* `x)`:

```
(gl::def-gl-rewrite logbitp-of-ash-minus1
  (implies (syntaxp (integerp n))
           (equal (logbitp n (ash x -1))
                  (logbitp (+ 1 (nfix n)) x))))
```

All of the `logbitp` test terms are generated twice, once for each call `(loghead 5 x)`, but the second time each one is recognized and the Boolean variable generated the first time is returned instead of generating a fresh one.

This strategy can be used to reason about more complex structures. For example, if a conjecture references an arbitrary record but only accesses a fixed set of fields and a fixed number of bits of each field, then it can generate Boolean variables as above for the fields, such as `(logbitp` *n* `(g` *field* `x))`.

2.3 Generating Term-level Counterexamples from Boolean Counterexamples

GL's method of proving a theorem is to obtain a Boolean formula representing the conclusion and then ask whether it is valid, either by examination of the canonical BDD representation or by querying a SAT solver to determine whether the negation of the AIG representation is satisfiable. When the result is valid (its negation is unsatisfiable), this proves the theorem, at least when all side obligations are met. But when it is not valid, a contradiction at the Boolean level does not directly yield a real counterexample to the theorem — instead, it gives a Boolean valuation to each of the terms that were assigned Boolean variables as described above.[1] Consider the following example:

```
(gl::def-gl-thm minus-logext-minus-loghead-is-logext-loghead
   :hyp t
   :concl (equal (- (logext 5 (- (loghead 5 x))))
```

[1]This problem does not exist in traditional GL, where all variables of a conjecture are given bit-level symbolic representations by the g-bindings, and no other Boolean variables are generated. A bit-level counterexample then directly yields valuations for the term variables by evaluating their symbolic representations under the counterexample assignment.

```
                (logext 5 (loghead 5 x)))
   :g-bindings nil
   :rule-classes nil)
```

For this conjecture, a counterexample produced by the SAT solver includes the following assignments for the `logbitp` terms that make up the `loghead` (as described in the previous section):

```
(logbitp 0 x) = NIL
(logbitp 1 x) = NIL
(logbitp 2 x) = NIL
(logbitp 3 x) = NIL
(logbitp 4 x) = T
```

We would like to translate the valuations for these terms into a value for `x` itself. In this case, it seems obvious that we should assign `x` the value 16, or -16, or any integer whose low 5 two's-complement bits are 16: there isn't a unique assignment satisfying the given Boolean valuation. Additionally, it may or may not even be possible to set the term variables such that the terms all have their assigned values. For example, a badly configured set of rewrite rules could result in `(logbitp 0 (ash x -1))` and `(logbitp 1 x)` being assigned independent Boolean values; if those values differ, then there is no `x` for which both valuations hold. Furthermore, since Boolean variables can be assigned to arbitrary terms, it could be an arbitrarily hard problem to simultaneously satisfy the Boolean valuations of the terms. So we are looking for a heuristic solution that works in practice, since a total solution doesn't exist.

Our solution is to allow the user to specify rewrite rules that describe how to update variables and subterms in response to assignments. Every variable starts out assigned to `NIL`, and assignments to terms involving the variables may cause that value to be updated if appropriate rules are in place. With no such rules, the above theorem fails to prove but the counterexample produced assigns `NIL` to `x`, which is a false counterexample. However, if we add the following rule, then it will instead assign 16 to `x`, which is a real counterexample:

```
(gl::def-glcp-ctrex-rewrite
  ((logbitp n i) v)
  (i (bitops::install-bit n (bool->bit v) i)))
```

This rule reads: "If a term matching `(logbitp n i)` is assigned value `v`, then instead assign `i` the evaluation of `(bitops::install-bit n (bool->bit v) i)`." With this rule in place GL correctly generates the counterexample `x = 16` for the above proof attempt. These rules do not incur any proof obligations, since they are not used for anything relevant to soundness. Note also that these operations are performed without guard checking, so that installing a bit into `NIL` does not cause an error.

Counterexample rewrite rules work together to resolve nested accesses. For example, we could have another rule for dealing with record accesses:

```
(gl::def-glcp-ctrex-rewrite
  ((g field rec) v)
  (rec (s field v rec)))
```

This rule can work in concert with the `logbitp` rule above to resolve assignments to terms such as `(logbitp 4 (g :fld x)) = t`. Suppose that before processing this assignment, `x` was bound to `nil`. The `logbitp` rule fires first and replaces our assignment with `(g :fld x) = (bitops::install-bit 4 (bool->bit t) (g :fld nil))`, which evaluates to 16. Then the rule for `g` applies, and results in the assignment `x = (s :fld 16 nil)`, which evaluates to `((:FLD . 16))`. This is then the provisional value for `x`, until it is updated by the resolution of another Boolean assignment.

Most counterexample rewrite rules essentially describe how to update an aggregate so that a particular field of that aggregate will have a given value: the `logbitp` rule above shows how to update a particular bit of an integer, and the `g` rule shows how to update a field in a record.

2.4 Adding Constraints among Generated Boolean Variables

When assigning Boolean variables to arbitrary term-level conditions, a pitfall is that by default all such variables are assumed to be independent. As noted in Section 2.3, this isn't guaranteed; it may be impossible for the Boolean valuation to be satisfied by an assignment to the term-level variables. The following form shows a proof attempt that fails due to a Boolean assignment that can't be realized by a term-level assignment:

```
(gl::def-gl-thm loghead-equal-12-when-integerp
  :hyp t
  :concl (equal (and (integerp x)
                     (equal (loghead 5 x) 12))
                (equal (loghead 5 x) 12))
  :g-bindings nil)
```

The Boolean formula given to the SAT solver for this conjecture has variables for `(integerp x)` and for `(logbitp` *n* `x)` for *n* from 0 to 4. The counterexample it returns has `(integerp x) = NIL` and the `logbitp`s set so that `(loghead 5 x) = 12`. But this is impossible because, e.g., `(logbitp 2 x) = NIL` if `x` is not an integer: in fact, `(logbitp` *n* `x)` implies `(integerp x)`. GL allows a kind of rule that encodes these sorts of constraints among Boolean conditions:

```
(gl::def-gl-boolean-constraint logbitp-implies-integerp
  :bindings ((bit0 (logbitp n x))
             (intp (integerp x)))
  :body (implies bit0 intp))
```

This form submits the following theorem:

```
(let ((bit0 (if (logbitp n x) t nil))
      (intp (if (integerp x) t nil)))
  (implies bit0 intp))
```

It additionally adds information about the constraint rule to a database that is used to track relationships among GL's generated Boolean variables. In this case it keeps track of Boolean variables added for terms matching `(logbitp n x)` and `(integerp x)`, and for each combination of these terms it evaluates the theorem body and adds the resulting Boolean formula to a list of constraints. When the theorem's final SAT query is constructed, all such constraints are included in the formula to be checked, preventing the false counterexamples that led to our proof failure above. With this constraint rule, the theorem proves.

The distinction between bindings and body in a Boolean constraint rule has important effects on how the rule is applied. All bindings must be matched to terms with existing Boolean variables before the rule applies. So in the version above, both `(logbitp n x)` and `(integerp x)` terms must be encountered as `if` tests before the rule fires. We can rewrite the rule as follows so that the constraint is added as soon as a term matching `(logbitp n x)` is encountered:

```
(gl::def-gl-boolean-constraint logbitp-implies-integerp-stronger
  :bindings ((bit0 (logbitp n x)))
  :body (implies bit0 (integerp x)))
```

Constraint rules can be used to perform a kind of forward reasoning. Suppose that we used terms such as

```
(logbitp 0 (ash (ash (... (ash x -1) ...) -1) -1))
```

as our normal form for bits extracted from a variable `x`, rather than `(logbitp` *n* `x)`; this occurs if we omit the rule `logbitp-of-ash-minus1` stated in Section 2.2. The following two rules then suffice to prove our theorem:

```
(gl::def-gl-boolean-constraint logbitp-implies-nonzero
  :bindings ((bit0 (logbitp n i)))
  :body (implies bit0 (and (integerp i) (not (equal 0 i)))))

(gl::def-gl-boolean-constraint nonzero-rsh-implies-nonzero
  :bindings ((iszero (equal 0 (ash i -1))))
  :body (implies (not iszero) (and (integerp i) (not (equal 0 i)))))
```

Whenever a `logbitp` term is derived from the `loghead` in our theorem, this triggers the first rule. The body of that rule is evaluated in an `if`-test-like context, so that `(integerp i)` and `(equal 0 i)` both are assigned Boolean variables if necessary. Then if `i` matched a nesting of `ash` terms, the addition of `(equal 0 i)` will lead to the second rule firing repeatedly until `i` unifies with the innermost variable. Together these set up a chain of implications so that ultimately, if the `logbitp` is true, it is known that the inner variable is an integer.

2.5 Rules to Unify Representations Between `if` Branches

One of the main strengths of GL is that it allows case splits that could otherwise lead to combinatorial blow-ups to be represented symbolically in Boolean function objects. As an abstract example, suppose each function `f`*n* in the term below produces a 10-bit natural:

```
(let* ((x1   (if (c1 x0) (f11 x0) (f12 x0)))
       (x2   (if (c2 x1) (f21 x1) (f22 x1)))
       (x3   (if (c3 x2) (f31 x2) (f32 x2)))
       (x4   (if (c4 x3) (f41 x3) (f42 x3)))
      ...)
    x10)
```

Proving this using ACL2 would likely require a case-split into 1024 cases, but those cases can all be represented simultaneously in the bits of one symbolic number. Each `if` term produces a number on each branch, which are merged into a common symbolic representation before continuing on with the next term. These potential case splits are then handled in the BDD engine or SAT solver rather than by explicit enumeration.

To extend this feature to term-level representations, GL allows rules to merge branches of `if`s that result in different term-level representations into a unified representation. An example for the theory of records:

```
(gl::def-gl-branch-merge merge-if-of-s
 (equal (if c (s k v rec1) rec2)
        (s k
           (if c v (g k rec2))
           (if c rec1 rec2))))
```

The left-hand side of this rule should always be an `if` with variables for the test and the else branch, but some function call for the `then` branch. It is applied to both branches symmetrically, so in this case if a call of `s` appears in the else but not then branch, it will still be applied (it effectively matches the same `if` with the test negated and branches swapped). This rule is particularly useful when the two branches of an `if` update different keys of a record. When using this rule it is best to also have rewrite rules in place to eliminate redundant sets of the same key, since otherwise setting the same key on both branches will cause the set of the key to be nested.

2.6 Function Call Shape Specifier

Shape specifiers, given in the `:g-bindings` argument to `def-gl-thm`, bind term variables to symbolic representations. Without the other term-level features discussed here, the only practical way to use GL was to provide such a binding for each variable, usually to a symbolic Boolean, a symbolic integer of fixed bit width, or a cons structure containing symbolic Booleans and/or integers. The Boolean variable generation feature discussed in Section 2.2 makes it possible to prove some practical theorems while leaving variables unbound (effectively binding them to term-level variables named after themselves). But when using term-level features it can still be advantageous to choose a BDD variable ordering or otherwise control the representation of theorem variables. The `g-call` shape specifier is a new feature that allows a theorem variable to be bound to a term-level function call. The difficulty of using such a construct is in proving that the shape specifier covers the required range of objects. If we have the hypothesis `(signed-byte-p 8 x)`, then it is easy to prove that an 8-bit integer shape specifier covers all needed inputs, but if the shape specifier instead is a function call then we need some knowledge about the function.

The coverage obligation for a given shape specifier is to show that for any target object satisfying some assumption (the theorem hypotheses, at the top level), there exists an environment under which the symbolic object generated from the shape specifier evaluates to the target object. For most kinds of shape specifiers, this can be determined syntactically: as noted above, an 8-bit integer shape specifier is known to cover all 8-bit integers, because every bit is required to be an independent Boolean variable. But to show that a function applied to some shape specifier arguments covers some object, we need to know what arguments to that function produce that object, i.e., we need an inverse function. The `g-call` shape specifier therefore contains not only the function and argument list of the call, but also a third field giving the inverse function, which must be a single-argument function symbol or lambda. Applied to a target object in the required coverage range, this function should produce an argument list on which the function produces that target object. We can then split the coverage obligation into two parts: the function applied to its inverse produces the target object, and the inverse is in the range of the argument shape specifiers.

To relieve the coverage obligation for the shape specifiers, GL generates a proof obligation term which it returns from the clause processor as a side goal. The proof obligation term is computed by the function `(shape-spec-oblig-term` *sspec target*`)`. When the given shape specifier *sspec* does not contain any `g-call` objects, the obligation is stated as `(shape-spec-obj-in-range` *sspec target*`)`, which syntactically determines whether the shape specifier covers the target. On a `g-call` shape specifier, `shape-spec-oblig-term` produces two obligations using the provided inverse term. If *sspec* is `(g-call` *fn args inv*`)`, the obligations are:

1. The function applied to the argument list produced by the inverse function on the target object equals the target object (here n is the length of *args* minus 1):

```
(let ((inv-args (inv target)))
  (equal (fn (nth 0 inv-args) ... (nth n inv-args)))
         target)
```

2. The shape specifiers of the argument list recursively cover the values produced by the inverse function when applied to the target object: the obligations showing this are generated by recursively calling `shape-spec-oblig-term` on the arguments and the corresponding `nth`s of the call of the inverse function.

Here is an example:

```
(defun plus (st dest src1 src2)
  (s dest (loghead 10 (+ (g src1 st) (g src2 st))) st))

(defun s-inverse (key obj)
  (list key (g key obj) obj))

(gl::def-gl-thm plus-c-a-b-correct
  :hyp (and (unsigned-byte-p 9 (g :a st))
            (unsigned-byte-p 9 (g :b st)))
  :concl (let* ((new-st (plus st :c :a :b))
                (a (g :a st))
                (b (g :b st))
                (new-c (g :c new-st)))
           (equal new-c (+ a b)))
  :g-bindings `((st ,(gl::g-call
                      's
                      (list :a
                            (gl::g-int 0 1 10)
                            (gl::g-call
                             's
                             (list :b
                                   (gl::g-int 10 1 10)
                                   (gl::g-var 'rest-of-st))
                             '(lambda (x) (s-inverse ':b x))))
                      '(lambda (x) (s-inverse ':a x)))))
  :cov-theory-add '(nth-const-of-cons
                    s-same-g
                    s-inverse))
```

The first two arguments in each `g-call` are the function name and argument list, and the third argument gives the inverse function. In the example, the variable `st` is bound to an object representing the following term, where *a* and *b* are 10-bit integer shape specifiers:

`(s :a` *a* `(s :b` *b* `rest-of-st))`

The inverse function for each `g-call` produces a list of three values, corresponding to the arguments *key*, *val*, *rec* of `s`. The key is chosen to match the key used by the call, the value is chosen to match the existing binding of the key in the record, and the record is just the input record itself. By the

record property `s-same-g`, `(equal (s a (g a r) r) r)`, this reduces to the input record, satisfying obligation 1.[2] The shape specifiers for the arguments `(key val rec)` also cover the values, satisfying Obligation 2: the keys are equal constants; the value shape specifiers are each 10-bit integers, which by the `unsigned-byte-p` hypotheses suffice to cover the specified components, and the `g-var` construct used as the innermost record can cover any object. The coverage proof completes with the help of the `:cov-theory-add` argument, which enables the listed rules during the proof of the coverage obligation.

3 Implementation Details

While an in-depth description of the implementation of these features is beyond the scope of this paper, we mention here some implementation details that may be relevant to users.

3.1 Priority Order of Function Call Reductions

As the GL symbolic interpreter crawls over a term, at each function call subterm it first interprets the arguments to the call, reducing them to symbolic objects (though possibly term-like ones). Then there are several reductions that may be used to resolve the function call to a symbolic object. These reductions are affected by the setting of some functions as uninterpreted, as first discussed in Section 2.1; these functions are stored in a table where their names are bound to `NIL` (interpreted), `T` (uninterpreted), or `:CONCRETE-ONLY` (uninterpreted, but allowed to be concretely evaluated). The reductions are listed here in the order they are considered.[3]

1. If the arguments are all syntactically concrete (implying they each have only one possible value), and the function is allowed to be concretely evaluated (its uninterpreted setting is not `T`), then the function is evaluated on the arguments' values using `magic-ev-fncall` and the resulting value is returned, properly quoted as a concrete object.
2. Rewrite rules are read from the function's `lemmas` property; any rules whose runes are listed in the `gl-rewrite-rules` table are attempted. If any such rule is successfully applied, the resulting term is symbolically interpreted in place of the function call.
3. If the function has a custom symbolic counterpart defined in the current clause processor [5] (see documentation topic gl::custom-symbolic-counterparts), it is run on the arguments and its value returned.
4. If the function is uninterpreted, a new function call object is returned containing the function name applied to the argument objects.
5. Otherwise, the body of the function is interpreted with the arguments bound to its formals. The particular definition used may be overridden; see documentation topic gl::alternate-definitions.

3.2 Handling of `if` Terms

There are two main parts of the handling of `if` terms: interpreting the test, and merging the branches when the test is not resolved to a constant value.

[2] The fact that we don't need to assume that the input value is a record is due to a special property of the records library, which goes to some pains to ensure that its functions work together logically without type assumptions.

[3] This is coded in the book "centaur/gl/glcp-templates.lisp" under `(defun interp-fncall ...)`; this template is expanded to an actual function definition when a new GL clause processor is created using `def-gl-clause-processor`.

First, the test term is symbolically interpreted, resulting in some symbolic object. That symbolic object is coerced to a Boolean function representation as follows:[4]

1. If it is a symbolic Boolean, extract its Boolean function.
2. If it is a symbolic number, return constant true.
3. If it is a cons, return constant true.
4. If it is syntactically concrete, return constant true its value is if non-`nil`, constant false if `nil`.
5. If it is an if-then-else, recursively extract the Boolean function of the test. If that function is syntactically constant true or constant false, return the Boolean function of the corresponding branch; else get the Boolean functions of both branches and return the Boolean if-then-else of the test and branches.
6. If it is a variable term, return its associated Boolean variable if it has one, else generate a fresh Boolean variable, record its association, and return it.
7. If it is a function call term matching `(not` *x*`)` or `(equal` *x* `nil)`, then recursively extract the Boolean function for *x* and return its negation.
8. If it is a function call term matching `(gl::gl-force-check-fn` *x strong dir*`)`, recursively extract the Boolean function for *x* and use satisfiability checking to try to reduce it to constant-true and/or constant-false; see the documentation of gl::gl-force-check.
9. If it is a function call term not matching any of the above, return its associated Boolean variable if it has one, else generate a fresh Boolean variable, record its association, update the constraint database, and return the variable.

If the resulting Boolean function is syntactically constant or known true or false under the path condition, then only one of the two branches needs to be followed, and no merging is necessary. Otherwise, each branch is symbolically interpreted with the path condition updated to reflect the required value of the test, and the results are merged. Merging works as follows:[5]

1. If both branch objects are equal, return that object.
2. If the *then* branch is a function call object (or a cons object, which is viewed as a call of `cons` here), try to rewrite the `if` with all branch merge rules stored under the function symbol. If any succeeds, proceed to symbolically interpret the resulting term.
3. Similarly if the *else* branch is a function call or cons object, try to rewrite the inverted `if` term `(if` ¬*test else then*`)` with branch merge rules.
4. If both branches are calls of the same function or both conses, recursively merge the corresponding arguments and interpret the application of the function to the merged arguments.
5. Otherwise, merge the two branch objects without further symbolic interpretation or rewriting, merging components of similar type and making `if` objects when necessary [5].

[4] Coded in "centaur/gl/glcp-templates.lisp" under `(defun simplify-if-test ...)`

[5] Coded in "centaur/gl/glcp-templates.lisp" under `(defun merge-branches ...)`

4 Record Example

We'll develop a useful theory here for reasoning about records in GL, under the assumption that keys will generally be concrete values; this won't work as well for memories where addresses are sometimes computed, for example.

To effectively reason about records, first it is important that the basic functions involved are uninterpreted. Here we use the `:concrete-only` option so that they can be concretely executed when necessary.

```
(gl::gl-set-uninterpreted g :concrete-only)
(gl::gl-set-uninterpreted s :concrete-only)
```

The most important rule for reasoning about records is this rule, which determines the value extracted by a lookup after an update:

```
(gl::def-gl-rewrite g-of-s-casesplit
  (equal (g k1 (s k2 v x))
         (if (equal k1 k2)
             v
           (g k1 x))))
```

It is often desirable to normalize the ordering of keys in nests of `s` operators, to avoid growing terms out of control. Here `general-concretep` and `general-concrete-obj` respectively check whether a symbolic object is constant and return its constant value. The third hypothesis of the following rule ensures that the rule does not loop and normalizes the ordering of keys to `<<` sorted order.

```
(gl::def-gl-rewrite s-of-s-normalize
  (implies (syntaxp (and (gl::general-concretep k1)
                         (gl::general-concretep k2)
                         (not (<< (gl::general-concrete-obj k1)
                                  (gl::general-concrete-obj k2)))))
           (equal (s k1 v1 (s k2 v2 x))
                  (if (equal k1 k2)
                      (s k1 v1 x)
                    (s k2 v2 (s k1 v1 x))))))
```

Other disciplines for organizing the record structure may perform better in certain contexts: it might be preferable to avoid sorting keys, or to choose a representation that allows a specialized structure such as a tree or fast alist to be used for record lookups.

To generate good counterexamples for theorems involving `g` and `s`, the following rule shows how to update a record so that the given key is bound to a particular value:

```
(gl::def-glcp-ctrex-rewrite
  ((g k x) v)
  (x (s k v x)))
```

The following rule shows how to merge branches in which one branch resulted in an `s` call:

```
(gl::def-gl-branch-merge merge-if-of-s
  (equal (if test (s k v x) y)
         (s k (if test v (g k y)) (if test x y))))
```

The above rules are generally sufficient to reason about records provided we only care about equality of certain fields. If we want to check equality of records, we need some rewriting tricks.

We use a helper function here that checks that two records are equal except possibly in their values on a particular list of keys. This is equivalent to the quantified function

```
(forall k
    (implies (not (member k lst))
             (equal (g k x) (g k y))))
```

but the following recursive formulation is easier to reason about:

```
(defun gs-equal-except (lst x y)
  (if (atom lst)
      (equal x y)
    (gs-equal-except (cdr lst)
                     (s (car lst) nil x)
                     (s (car lst) nil y))))

(gl::gl-set-uninterpreted gs-equal-except)
```

We can use this function to unwind equalities of `s` operations to a conjunction of equalities between the stored values for each key set. This theorem starts the process (the theorem with swapped LHS arguments is also needed):

```
(gl::def-gl-rewrite equal-of-s
  (equal (equal (s k v x) y)
         (and (equal v (g k y))
              (gs-equal-except (list k) x y))))
```

This theorem (and the similar theorem with swapped arguments to the LHS `gs-equal-except`) unrolls the rest of the way until we reach the original records at the beginning of the `s` nest:

```
(gl::def-gl-rewrite gs-equal-except-of-s
  (equal (gs-equal-except lst (s k v x) y)
         (if (member k lst)
             (gs-equal-except lst x y)
           (and (equal v (g k y))
                (gs-equal-except (cons k lst) x y)))))
```

Finally, this theorem accounts for the base case in which we have no more `s` calls and just compare the original records directly. This suffices in the common case where the original record is the same exact object on both sides of the comparison. The syntaxp check here ensures that we don't rewrite our `equal` hypothesis with `equal-of-s` and cause a loop.

```
(gl::def-gl-rewrite gs-equal-except-of-s-base-case
  (implies (and (syntaxp (and (not (and (consp x)
                                        (eq (gl::tag x) :g-apply)
                                        (eq (gl::g-apply->fn x) 's)))
                              (not (and (consp y)
                                        (eq (gl::tag y) :g-apply)
                                        (eq (gl::g-apply->fn y) 's)))))
                (equal x y))
           (equal (gs-equal-except lst x y) t)))
```

This rewriting strategy isn't complete for proving equality or inequality between records. For example, if we have a comparison of two different variables and an assumption that they differ on some key, we don't resolve the equality to false. However, it does suffice for the common usage pattern of showing that a machine model running some program produces the same final state as a specification.

5 Application

The GL features discussed here have been used in proofs about microcode routines at Centaur Technology [2]. The proof framework was based on specifying the updates to the machine state computed by small code blocks, then composing these blocks together to specify the resulting machine state from a whole microcode routine. The correctness proof for each code block would prove that if the starting state satisfied certain conditions – such as the program counter having the correct value, or certain constraints holding of data elements – that running the machine model for some number of steps produced a state equal to that produced by the specification. Don't-care fields could be "wormhole abstracted" [3], essentially specifying their value as whatever was produced by the implementation.

The machine model used in the microcode proofs was a stobj (for best concrete execution performance) but was modeled logically as a nested record structure in which field accessors always fixed the stored values to the appropriate type to match what was stored in the stobj (see the documentation for defrstobj). A GL term-level theory similar to the one described in Section 4 was used to handle the accesses and updates, and Boolean variable generation was used to generate representations for fields of the state, which were mostly fixed-width natural numbers.

A code block proof could be attempted either using GL or not; in some cases the traditional ACL2 proof procedure was preferable. GL was successfully used on basic code blocks of up to about 30 instructions, and also for some block composition theorems.

6 Soundness Proof

The GL clause processor is verified as a sound extension to ACL2. The symbolic interpreter at its core is a 22-function mutual recursion, consisting of four functions responsible for different steps in interpreting a term, five functions to implement the conditional rewriter, ten dealing with processing of `if` terms, two responsible for adding Boolean variable constraints, and one that interprets a list of terms. To reason about this mutual recursion, we used the utilities from community book "tools/flag.lisp" and a wrapper macro to support proving similar theorems about all the functions in the mutual recursion. To get the proofs about this mutual recursion to perform acceptably, we tried to make each function as simple as possible, which sometimes involved splitting the functions into more mutually-recursive parts.

Previous to the addition of the term-level features, the main theorem about the interpreter was that the evaluation of the output symbolic object under some environment was equal to the evaluation of the input term under the alist formed by evaluating each of the symbolic objects in the input symbolic object bindings. The statement, greatly simplified, looked like this:

```
(implies (and (bool-function-eval pathcond env)
              ...)
         (equal (sym-object-eval (gl-interpret-term x bindings pathcond) env)
                (term-eval x (sym-object-alist-eval bindings env))))
```

Of the new features discussed here, the only one that added significant complexity or difficulty to this theorem was the generation of fresh Boolean variables (Section 2.2). Rewriting and `if` branch merging

rules added functions to the mutual recursion, but didn't require us to introduce any new concepts into the proofs.

The Boolean variable generation feature complicates the relationship between the input bindings and output symbolic object in the theorem above because the output term may reference Boolean variables that were created during the course of the symbolic interpretation. The meanings of these new Boolean variables are stored in a database that associates the generated variables (represented as natural numbers, generated in increasing order) with the symbolic objects they were generated to represent. The symbolic interpreter takes a Boolean variable database as input and returns it, possibly extended with new variable/object correspondences, as output. Existing correspondences in the database are not changed during symbolic interpretation.

In order to prove a correspondence between output symbolic object and input term and bindings similar to the one above, we need some facts about the set of Boolean variables on which each object depends:

- The symbolic object returned by the interpreter may only depend on Boolean variables present in the input bindings or bound in the Boolean variable database.
- Symbolic objects bound to Boolean variables in the Boolean variable database are strictly ordered in their dependencies: the object bound to a given Boolean variable may only depend on lower-numbered variables.

We also need to assume:

- The symbolic objects in the input bindings do not depend on any Boolean variable numbers greater than the maximum currently present in the Boolean variable database.

These show that adding a new variable/object correspondence to the database does not affect the meaning of existing objects. Finally, an additional assumption is needed about the environment under which the output object and input bindings are evaluated:

- The environment is consistent with the Boolean variable database: that is, for each Boolean variable/object binding (v, o) in the database, the binding of v in the environment must agree with the truth value of the evaluation of o under the environment.

With these assumptions, we can prove a correspondence between the output symbolic object and input term and bindings similar to the one above. However, we must also show that environments satisfying the above assumption exist; in particular, given an environment under which the input shape specifiers evaluate to a particular assignment of theorem variables, we must show that there exists an environment that preserves this evaluation and is consistent with the Boolean variable database. We do this by iteratively mapping the Boolean variables from the database to the evaluations of their corresponding objects, in order from the lowest-numbered. The result, we prove, is an environment that preserves the evaluation of the shape specifiers and is consistent with the database. This allows us to argue that if the Boolean function representation of the theorem's conclusion is valid, then the theorem is true: given an assignment to the theorem's variables under which the theorem is false:

1. this assignment satisfies the hypothesis, so by coverage there exists an environment under which the shape specifiers evaluate to that assignment;
2. we can extend that environment to be consistent with the Boolean variable database while preserving the evaluation of the shape specifiers;
3. therefore the result of symbolic simulation of the conclusion under that environment must agree with the evaluation of the theorem under that assignment;
4. but since the symbolic simulation result is a valid Boolean function, the theorem is true.

7 Future Extensions

7.1 Fixtypes Support

The FTY library (see documentation for FTY) provides automation for defining (mutually-)recursive sum-of-products, product, list, and alist types that support a fixing discipline that avoids the need to assume that variables are well-typed when proving theorems [7]. In this fixing discipline, all types have corresponding fixing functions, which logically fix their argument to the corresponding type but are essentially free to execute in guard-verified code because the inputs are known to already be of the correct type. Unfortunately, this fixing discipline could potentially have a very bad performance impact in GL, where the logical version of the code is used and guards are ignored. This also affects performance when calling discipline-compliant functions on concrete values, because each call from the symbolic interpreter must first check its guard.

For cases where the fixing discipline becomes a practical problem, it may be a better approach to treat the types involved as abstract datatypes. The FTY library already provides rewrite rules for the ACL2 rewriter that do this, and some of these could be used directly in GL. Other rules necessary for GL to effectively reason about an FTY type could be generated automatically. The FTY type definition events leave behind information in ACL2 tables that fully describe the types. It would be reasonable to use this information to generate a GL term-level theory for a given FTY type.

7.2 Feature Parity with the ACL2 Rewriter

GL's rewriter is missing several important features of ACL2's simplifier. We list these in order from most to least likely to be worth the implementation effort.

- Bind-free and binding hypotheses are useful for effective programming of the rewriter and would be relatively easy to implement. The primary burden would be to modify the correctness proof of the rewriter/symbolic interpreter; we would need to show that these hypotheses only extend the current unifying substitution with free variables, and that such extensions to the substitution are allowable when applying a rewrite rule.
- Congruence-based reasoning is also somewhat likely to be useful for programming the rewriter, and is partly implemented already. GL already distinguishes between `equal` and `iff` contexts and the rewriter variable that makes this distinction already has been extended to handle other equivalences; the correctness proof also accounts for arbitrary equivalences. The parts yet to be implemented are the parsing and checking of congruence, refinement, and equivalence rules so that we can soundly use these theorems, and the application of congruences to the equivalence context when beginning to interpret a new subterm.
- Free-variable matching hypotheses are somewhat of a mismatch to GL's reasoning paradigm. Whereas ACL2 has a clause from which it derives a type-alist that stores its assumptions, GL only has a path condition. The path condition typically stores assumptions that are Boolean formulas rather than terms that are known true. Nevertheless, we could scan the path condition for known-true terms that unify with the current hypothesis under the current substitution. The proof obligations would be the same as for the bind-free and binding hypotheses, so only this implementation work would be necessary.
- Forward-chaining is an ACL2 feature that also doesn't fit cleanly into GL's paradigm, for similar reasons as for free-variable matching hypotheses. We could allow forward reasoning when adding

an `if` test to the path condition, which would require some non-insurmountable amount of new implementation work. The Boolean variable constraint system (Section 2.4) already allows forward reasoning after a fashion, but this doesn't affect rewriting directly since the constraints generated are currently only used in Boolean proofs.

7.3 Debugging Support

GL currently does not have any support beyond ACL2's `trace$` utility for debugging failed term-level proofs. While `trace$` is extremely useful to a user knowledgeable enough about implementation details to know what functions to trace, targeted debugging tools such as ACL2's break-rewrite would be a dramatic usability improvement.

8 Conclusion

This paper describes extensions to the GL bitblasting framework that allow it to deal in abstract terms, rather than a limited value-like representation. The GL framework is available under a permissive license in the ACL2 community books.

Acknowledgements

We would like to acknowledge Jared Davis for helpful conversations during the implementation of the features discussed, and Shilpi Goel for testing out some of these features during her dissertation research. We also thank the reviewers for helpful comments.

References

[1] ACL2 Community (accessed January, 2017): *ACL2+Books Documentation*. Available at `http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/index.html`.

[2] Jared Davis, Anna Slobodova & Sol Swords (2014): *Microcode Verification–Another Piece of the Microprocessor Verification Puzzle*. In: *International Conference on Interactive Theorem Proving*, Springer, pp. 1–16, doi:10.1007/978-3-319-08970-6_1.

[3] David S Hardin, Eric W Smith & William D Young (2006): *A robust machine code proof framework for highly secure applications*. In: *Proceedings of the Sixth International Workshop on the ACL2 Theorem Prover and its Applications*, pp. 11–20, doi:10.1145/1217975.1217978.

[4] Matt Kaufmann & Rob Sumners (2002): *Efficient rewriting of operations on finite structures in ACL2*. In: *Proceedings of the 3rd International Workshop on the ACL2 Theorem Prover and its Applications*, pp. 141–150.

[5] Sol Swords (2010): *A Verified Framework for Symbolic Execution in the ACL2 Theorem Prover*. Ph.D. thesis, Department of Computer Sciences, The University of Texas at Austin. Available at `http://hdl.handle.net/2152/ETD-UT-2010-12-2210`.

[6] Sol Swords and Jared Davis (2011): *Bit-Blasting ACL2 Theorems*. In: *Proceedings 10th International Workshop on the ACL2 Theorem Prover and its Applications, ACL2 2011, Austin, Texas, USA, November 3-4, 2011.*, pp. 84–102, doi:10.4204/EPTCS.70.7.

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

Extended Abstract: Formal Specification and Verification of the FM9001 Microprocessor Using the DE System

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

We present our work on formally specifying and mechanically verifying the correctness of the FM9001 microprocessor design using the DE system. The FM9001 microprocessor design was originally specified and verified in the Nqthm logic by Brock and Hunt using the DUAL-EVAL system, the precursor of DE. The main challenge is that ACL2 does not support Nqthm's shell principle, which is used in the original work to formalize the memory model. Instead we represent a memory cell as a proper list of two elements in which the first element is a flag specifying the memory type of the cell, and the second element is a four-valued vector representing the value of that memory cell. This memory representation does not affect the proof strategy for FM9001 created in the previous work, except for establishing the monotonicity property of a netlist's module, which is part of the FM9001 verification procedure. The reason is because the state approximation notion is changed under our proposed representation of the memory model. By modifying the state approximation definition, we successfully prove that the monotonicity property for a module still holds using stricter hypotheses, and finally prove that the FM9001 microprocessor design is correct with respect to a behavioral specification.

1 Overview

The FM9001 microprocessor design was originally specified and verified in the Nqthm logic by Brock and Hunt using the DUAL-EVAL system [2]. We attempt to re-specify and re-verify the FM9001 netlist in the ACL2 logic using the DE system. The DE language is similar in spirit and syntax to the DUAL-EVAL language, but they are different in a number of subtle aspects [3]. In addition, along with our work on the FM9001 specification and verification, we also verify guards for the DE system.

Our verification of the FM9001 microprocessor basically follows the same approach as presented by Brock and Hunt. We basically follow the same proof strategy as done in the previous work. However, there are some lemmas for which we have to come up with a different strategy to prove since the previous strategy does not work in ACL2.

As in the previous work, we heavily use macros for the purpose of automating our framework. The problem is that we are unable to reuse the macros from the previous work since they were defined in Common Lisp. They used features that are not supported in ACL2. Hence we have to define our own macros in ACL2, which requires a fair amount of effort.

The main challenge in our verification of FM9001 is that we are unable to represent the memory model for FM9001 using the *shell principle* as done in the previous work, since there is no such principle in ACL2. Specifically, the FM9001 memory was modeled using tree structures, with values stored at the tips. It was modeled in the Nqthm logic by using the shell principle to introduce three shells: the shell constructor ROM tags read-only locations of the memory, while the shell constructor RAM tags read-write locations and the shell constructor STUB represents "unimplemented" portions. Each

Warren Hunt Jr. and Anna Slobodova (Eds.):
ACL2 Theorem Prover and its Applications
EPTCS 249, 2017, pp. 112–114, doi:10.4204/EPTCS.249.8

instance of the memory parts includes a value, which is returned when that memory location is read. As mentioned, we are unable to represent the memory model in ACL2 using the shell principle. Instead we represent a memory cell as a proper list of two elements in which the first element is a flag specifying the memory type of the cell (i.e., ROM, or RAM, or STUB), and the second element is the value of that memory cell. This memory representation does not affect the proof strategy for FM9001 created in the previous work, except for establishing the *monotonicity property* of a netlist's module, which is part of the FM9001 verification procedure. The reason is because the *state approximation* notion is changed under our proposed representation of the memory model. Below is the ACL2 version of the state approximation definition introduced in the previous work.

```
(defun s-approx (s1 s2)
  (cond ((or (consp s1) (consp s2))                              // (1)
         (if (consp s1) (if (consp s2)
                            (and (s-approx (car s1) (car s2))
                                 (s-approx (cdr s1) (cdr s2)))
                          nil) nil))
        ((or (ramp s1) (ramp s2)) ...)                           // (2)
        ((or (romp s1) (romp s2)) ...)                           // (3)
        ((or (stubp s1) (stubp s2)) ...)                         // (4)
        (t ...)))                                                // (5)
```

Since our model constructs memory cells as CONSES; cases (2), (3), and (4) in the above definition will never be satisfied. They are all subsumed in case (1). We are now forced to change the state approximation definition above by rearranging the order of cases to (2), (3), (4), (1), and (5). We also need the following property in order to establish the monotonicity property for a module.

```
(implies (s-approx s1 s2)
         (s-approx (cdr s1) (cdr s2)))
```

This property holds when we impose a constraint on the value of each memory cell that it must be a *four-valued vector*. This constraint does not change the correctness proofs for FM9001 because the FM9001 specification enforces a restriction that only *bit vectors* are stored in memory. After obtaining the above property, we then manage to establish the monotonicity property for a module using stricter hypotheses. In particular, the monotonicity property now requires the structure of the module's state and the structure of the netlist containing the module to be well-formed.

2 Conclusion

While our modeling of the memory model for FM9001 is not the same as in the previous work, we successfully verify the correctness the FM9001 microprocessor design. This work provides a library of verified hardware circuit generators that can be applied when reasoning about the synthesis of hardware circuits. This work is also a contribution to ACL2 for two reasons. First, it moves into the ACL2 regression suite one of the most important theorems proved by Nqthm. Second, it is the first step toward porting the entire Computational Logic verified stack [1, 4] from Nqthm to ACL2. We hope others will take the subsequent steps.

References

[1] W. R. Bevier, W. A. Hunt, Jr., J S. Moore & W. D. Young (1989): *Special Issue on System Verification*. Journal of Automated Reasoning 5(4), pp. 409–530.

[2] B. Brock & W. Hunt (1997): *The DUAL-EVAL Hardware Description Language and Its Use in the Formal Specification and Verification of the FM9001 Microprocessor*. In: Journal of Formal Methods in System Design, 11, Kluwer Academic Publishers, pp. 71–104, doi:10.1023/A:1008685826293.

[3] W. Hunt (2000): *The DE Language*. In M. Kaufmann, P. Manolios & J S. Moore, editors: Computer-Aided Reasoning: ACL2 Case Studies, chapter 10, Springer US, pp. 151–166, doi:10.1007/978-1-4757-3188-0_10.

[4] J S. Moore (1996): *Piton: A Mechanically Verified Assembly-Level Language*. Automated Reasoning Series, Kluwer Academic Publishers.