EPTCS 280

Proceedings of the

15th International Workshop on the ACL2 Theorem Prover and Its Applications

Austin, Texas, USA, November 5-6, 2018

Edited by: Shilpi Goel and Matt Kaufmann

Published: 29th October 2018
DOI: 10.4204/EPTCS.280
ISSN: 2075-2180
Open Publishing Association

Table of Contents

Preface

This volume contains the proceedings of the Fifteenth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2018), a two-day workshop held in Austin, Texas, USA, on November 5-6, 2018, immediately after FMCAD (`http://www.fmcad.org/FMCAD18`), on the University of Texas campus. ACL2 workshops occur at approximately 18-month intervals, and they provide a major technical forum for users of the ACL2 theorem proving system to present research related to the ACL2 theorem prover and its applications.

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

The proceedings of ACL2-2018 include eleven long papers and two extended abstracts. Each submission received three reviews. The workshop also included several "rump session" talks — short unpublished presentations that discussed ongoing research — as well as three invited talks:

- *Small Team, Short Ramp, Industrial-scale Problems: Some Experiments in the Practicum of Interactive Theorem Proving* — Sandip Ray (University of Florida)
- *Creating Formal Specifications of the Arm Processor Architecture* — Alastair Reid (ARM)
- *Building Blocks for RTL Verification in ACL2* — Sol Swords (Centaur Technology)

This workshop would not have been possible without the support of a large number of people. We thank our generous sponsors, Centaur Technology and Oracle Corporation. We thank those who contributed papers, as well as the members of the Program Committee for providing careful reviews and for engaging in lively discussion. We thank the members of the Steering Committee for their help and guidance. We are grateful to the invited speakers for sharing their insights. We thank The University of Texas (UT Austin), Warren Hunt, Jr., and Lindy Aleshire for helping us to get a venue for our workshop. We thank EasyChair, EPTCS, arXiv, and RegOnline for providing assistance with the logistics of the workshop. Thanks are also due to David Rager for helping us administer registration. We are also grateful for support of our time spent on workshop preparations from Centaur Technology, UT Austin, the Kestrel Institute, and DARPA under Contract Nos. FA8650-17-1-7704 and FA8750-15-C-0007. Last, but not least, we thank Mertcan Temel for his assistance with local arrangements. It has been a pleasure and an honor to chair this event.

Shilpi Goel and Matt Kaufmann
November, 2018

ACL2 2018 Program Committee

Harsh Chamarthi	Intel
Alessandro Coglio	Kestrel Institute
Jared Davis	Apple
Ruben Gamboa	University of Wyoming
Shilpi Goel	Centaur Technology
Dave Greve	Rockwell-Collins
Warren Hunt	The University of Texas
Sebastiaan Joosten	University of Twente
Matt Kaufmann	The University of Texas
John O'Leary	Intel
Grant Passmore	Aesthetic Integration
David Rager	Oracle
Sandip Ray	University of Florida
David Russinoff	ARM
Julien Schmaltz	Eindhoven University of Technology
Anna Slobodova	Centaur Technology
Eric Smith	Kestrel Institute
Sol Swords	Centaur Technology

A Simple Java Code Generator for ACL2 Based on a Deep Embedding of ACL2 in Java

Alessandro Coglio
Kestrel Institute
`http://www.kestrel.edu`

AIJ (**A**CL2 **I**n **J**ava) is a deep embedding in Java of an executable, side-effect-free, non-stobj-accessing subset of the ACL2 language without guards. ATJ (**A**CL2 **T**o **J**ava) is a simple Java code generator that turns ACL2 functions into AIJ representations that are evaluated by the AIJ interpreter. AIJ and ATJ enable possibly verified ACL2 code to run as, and interoperate with, Java code, without much of the ACL2 framework or any of the Lisp runtime. The current speed of the resulting Java code may be adequate to some applications.

1 Motivation and Contributions

A benefit of writing code in a theorem prover is the ability to prove properties about it, such as the satisfaction of requirements specifications. A facility to generate code in one or more programming languages from an executable subset of the prover's logical language enables the possibly verified code to run as, and interoperate with, code written in those programming languages. Assuming the correctness of code generation (whose verification is a separable problem, akin to compilation verification) the properties proved about the original code carry over to the generated code.

The ACL2 theorem provers's tight integration with the underlying Lisp platform enables the executable subset of the ACL2 logical language to run readily and efficiently as Lisp, without the need for explicit code generation facilities. Nonetheless, some situations may call for running ACL2 code in other programming languages: specifically, when the ACL2 code must interoperate with external code in those programming languages in a more integrated and efficient way than afforded by inter-language communication via foreign function interfaces [4, 9], or by inter-process communication with the ACL2/Lisp runtime via mechanisms like the ACL2 Bridge [2, `:doc bridge`]. Using Lisp implementations written in the target programming languages [1] involves not only porting ACL2 to them, but also including much more runtime code than necessary for the target applications. Compilers from Lisp to the target programming languages may need changes or wrappers, because executable ACL2 is not quite a subset of Lisp; furthermore, the ability to compile non-ACL2 Lisp code is an unnecessary complication as far as ACL2 compilation is concerned, making potential verification harder.

The work described in this paper contributes to the goal of running ACL2 code in other programming languages in the integrated manner described above:

- ATJ (**A**CL2 **T**o **J**ava) is a Java code generator for ACL2. ATJ translates executable, side-effect-free, non-stobj-accessing ACL2 functions, without their guards, into Java. It does so in a simple way, by turning the functions into deeply embedded Java representations that are executed by an ACL2 evaluator written in Java.
- AIJ (**A**CL2 **I**n **J**ava) is a deep embedding in Java of an executable, side-effect-free, non-stobj-accessing subset of the ACL2 language without guards. AIJ consists of (i) a Java representation of the ACL2 values, terms, and environment, (ii) a Java implementation of the ACL2 primitive

functions, and (iii) an ACL2 evaluator written in Java. AIJ executes the deeply embedded Java representations of ACL2 functions generated by ATJ. AIJ is of independent interest and can be used without ATJ.

The ACL2 language subset supported by ATJ and AIJ includes all the values, all the primitive functions, and many functions with raw Lisp code—see Section 2 for details on these two kinds of functions.

The initial implementation of AIJ favored assurance over efficiency: it was quite simple, to reduce the chance of errors and facilitate its potential verification, but it was also quite slow. The careful introduction of a few optimizations, which do not significantly complicate the code but provide large speed-ups, makes the speed of the current implementation arguably adequate for some applications; see Section 5. Furthermore, the code is amenable to additional planned optimizations; see Section 6.

2 Background: The Evaluation Semantics of ACL2

ACL2 has a precisely defined logical semantics [11], expressed in terms of syntax, axioms, and inference rules, similarly to logic textbooks and other theorem provers. This logical semantics applies to logic-mode functions, not program-mode functions. Guards are not part of the logic, but engender proof obligations in the logic when guard verification is attempted.

ACL2 also has a documented evaluation semantics [2, `:doc evaluation`], which could be formalized in terms of syntax, values, states, steps, errors, etc., as is customary for programming languages. This evaluation semantics applies to both logic-mode and program-mode functions. Guards affect the evaluation semantics, based on guard-checking settings. Even non-executable functions (e.g. introduced via `defchoose` or `defun-nx`) degenerately have an evaluation semantics, because they do yield error results when called; however, the following discussion focuses on executable functions.

Most logic-mode functions have definitions that specify both their logical and their evaluation semantics: for the former, the definitions are logically conservative axioms; for the latter, the definitions provide "instructions" for evaluating calls of the function. For a defined logic-mode function, the relationship between the two semantics is that, roughly speaking, evaluating a call of the function yields, in a finite number of steps, the unique result value that, with the argument values, satisfies the function's defining axiom—the actual relationship is slightly more complicated, as it may involve guard checking.

The primitive functions [2, `:doc primitive`] are in logic mode and have no definitions; they are all built-in. Examples are `equal`, `if`, `cons`, `car`, and `binary-+`. Their logical semantics is specified by axioms of the ACL2 logic. Their evaluation semantics is specified by raw Lisp code (under the hood). The relationship between the two semantics is as in the above paragraph, with the slight complication that `pkg-witness` and `pkg-imports` yield error results when called on unknown package names. The evaluation of calls of `if` is non-strict, as is customary.

Most program-mode functions have definitions that specify their evaluation semantics, similarly to the non-primitive logic-mode functions discussed above. Their definitions specify no logical semantics.

The logic-mode functions listed in the global variable `logic-fns-with-raw-code` have a logical semantics specified by their ACL2 definitions, but an evaluation semantics specified by raw Lisp code. (They are disjoint from the primitive functions, which have no definitions.) For some of these functions, e.g. `len`, the raw Lisp code just makes them run faster but is otherwise functionally equivalent to the ACL2 definitions. Others have side effects, carried out by their raw Lisp code but not reflected in their ACL2 definitions. For example, `hard-error` prints a message on the screen and immediately terminates execution, unwinding the call stack. As another example, `fmt-to-comment-window` prints a message on the screen, returning `nil` and continuing execution. But the ACL2 definitions of both of these example functions just return `nil`.

The program-mode functions listed in the global variable `program-fns-with-raw-code` have an evaluation semantics specified by raw Lisp code. Their ACL2 definitions appear to have no actual use.

Since stobjs [2, `:doc stobj`] are destructively updated, functions that manipulate stobjs may have side effects as well—namely, the destructive updates. Because of single-threadedness, these side effects are invisible in the end-to-end input/output evaluation of these functions; however, they may be visible in some formulations of the evaluation semantics, such as ones that comprehend interrupts, for which updating a record field in place involves different steps than constructing a new record value with a changed field. The built-in `state` stobj [2, `:doc state`] is "linked" to external entities, e.g. the file system of the underlying machine. Thus, functions that manipulate `state` may have side effects on these external entities. For example, `princ$` (a member of `logic-fns-with-raw-code`) writes to the stream associated with the output channel argument, and affects the file system.

The fact that the side effects of the evaluation semantics are not reflected in the logical semantics is a design choice that makes the language more practical for programming while retaining the ability to prove theorems. But when generating Java or other code, these side effects should be taken into consideration: for instance, turning `hard-error` and `fmt-to-comment-window` into Java code that returns (a representation of) `nil`, would be incorrect or at least undesired. As an aside, a similar issue applies to the use of APT transformations [3]: for instance, using the `simplify` transformation [6] to turn calls of `hard-error` into `nil`, while logically correct and within `simplify`'s stipulations, may be undesired or unexpected.

3 AIJ: The Deep Embedding

AIJ is a Java package whose public classes and methods provide an API to (i) build and unbuild representations of ACL2 values,[1] (ii) build representations of ACL2 terms and of an ACL2 environment, and (iii) evaluate calls of ACL2 primitive and defined functions, without checking guards. By construction, the ACL2 code represented and evaluated by AIJ is executable, has no side effects, does not access stobjs, and has no guards.

AIJ consists of a few thousand lines of Java code (including blank and comment lines), thoroughly documented with Javadoc comments. The implementation takes advantage of object-oriented features like encapsulation, polymorphism, and dynamic dispatch.

The Java classes that form AIJ are shown in the simplified UML class diagram in Figure 1 and described in the following subsections. Each class is depicted as a box containing its name.[2] Abstract classes have italicized names. Public classes have names preceded by +, while package-private classes have names preceded by ~. Inheritance ('is a') relationships are indicated by lines with hollow triangular tips. Composition ('part of') relationships are indicated by lines with solid rhomboidal tips, annotated with the names of the containing class instances' fields that store the contained class instances, and with the multiplicity of the contained instances for each containing instance ('0..*' means 'zero or more').

[1] When talking about AIJ, this paper calls 'build' and 'unbuild' what is often called 'construct' and 'destruct' in functional programming, because in object-oriented programming the latter terms may imply object allocation and deallocation, which is not necessarily what the AIJ API does.

[2] In AIJ's actual code, each class name is prefixed with '`Acl2`' (e.g. `Acl2Value`), so that external code can reference these classes unambiguously without AIJ's package name `edu.kestrel.acl2.aij`. This paper omits the prefix for brevity, and uses fully qualified names for the Java standard classes to avoid ambiguities, e.g. `java.lang.String` is the Java standard string class, as distinguished from `String` in Figure 1.

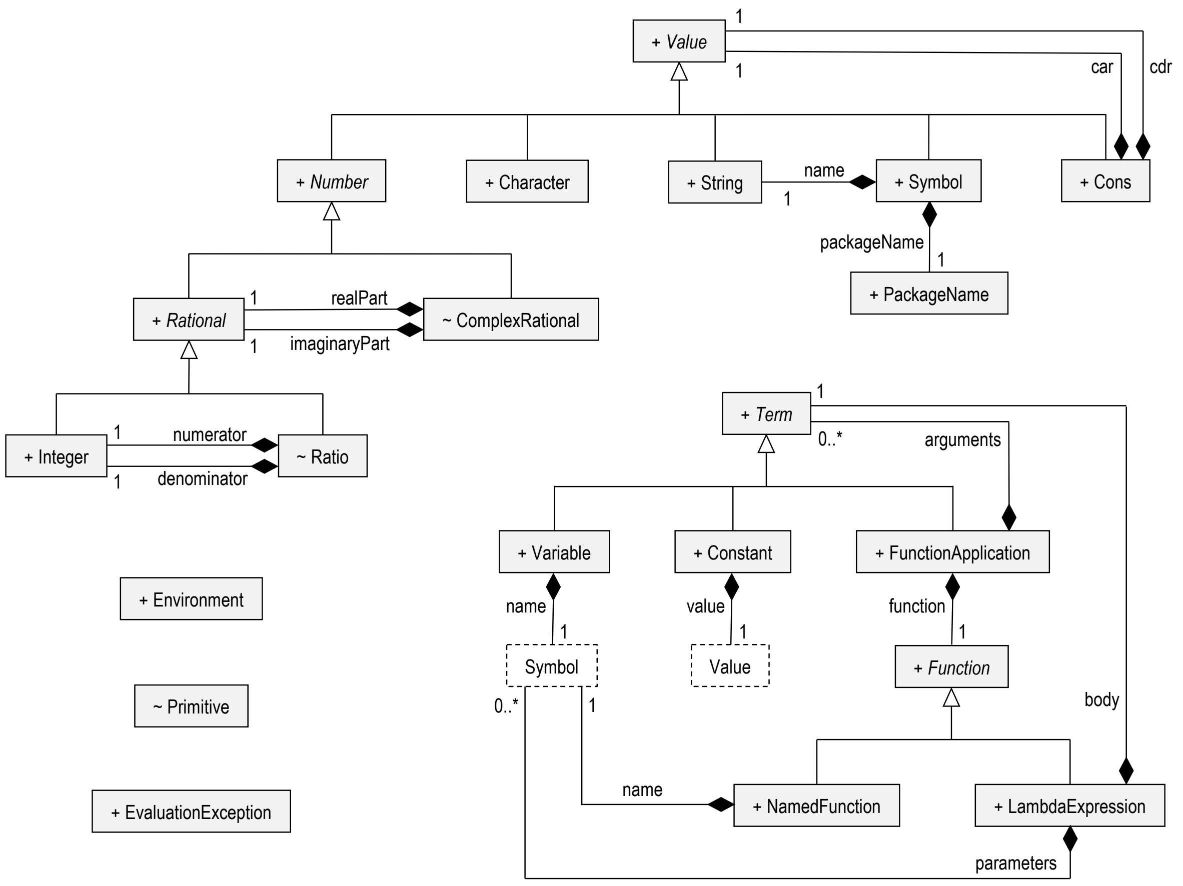


Figure 1: Simplified UML class diagram for AIJ.

The dashed boxes are just replicas to avoid clutter. This UML class diagram is simplified because the class boxes do not contain fields and methods.

3.1 Values

The set of values of the ACL2 evaluation semantics is the union of the sets depicted in Figure 2: (i) integers, recognized by `integerp`; (ii) ratios, i.e. rationals that are not integers, with no built-in recognizer;[3] (iii) complex rationals, recognized by `complex-rationalp`; (iv) characters, recognized by `characterp`; (v) strings, recognized by `stringp`; (vi) symbols, recognized by `symbolp`; and (vii) `cons` pairs, recognized by `consp`. Integers and ratios form the rationals, recognized by `rationalp`. Rationals and complex rationals form the Gaussian rationals, which are all the numbers in ACL2, recognized by `acl2-numberp`.[4] The logical semantics of ACL2 allows additional values called ‘bad atoms’, and consequently `cons` pairs that may contain them directly or indirectly; however, such values cannot be constructed in evaluation.

AIJ represents ACL2 values as immutable objects of `Value` and its subclasses in Figure 1. Each

[3]The term ‘ratio’ is used in the Common Lisp specification [14, Section 2.1.2].

[4]This discussion does not apply to ACL2(r).

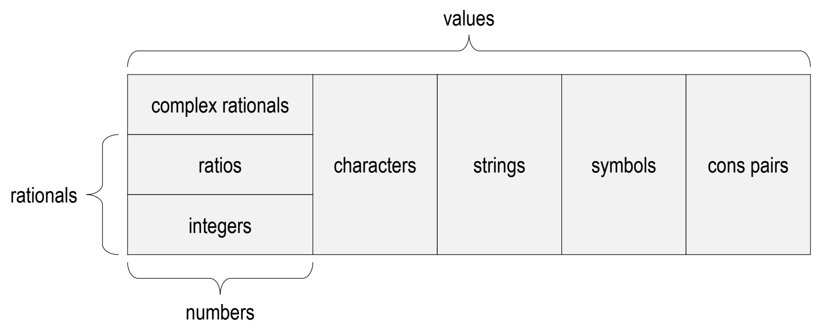


Figure 2: Values of the ACL2 evaluation semantics.

such class corresponds to a set in Figure 2. The subset relationships in Figure 2 match the inheritance relationships in Figure 1. The sets of values that are unions of other sets of values correspond to abstract classes; the other sets correspond to concrete classes. All these classes are public, except for the package-private ones for ratios and complex rationals: ratios and complex rationals are built indirectly via AIJ's API, by building rationals that are not integers and numbers that are not rationals.

The information about the represented ACL2 values is stored in fields of the non-abstract classes. `Integer` stores the numeric value as a `java.math.BigInteger`. `Ratio` stores the numerator and denominator as `Integers`, in reduced form (i.e. their greatest common divisor is 1 and the denominator is greater than 1). `ComplexRational` stores the real and imaginary parts as `Rationals`. `Character` stores the 8-bit code of the character as a `char` below 256. `String` stores the codes and order of the characters as a `java.lang.String` whose `chars` are all below 256. `Symbol` stores the symbol's package name as a `PackageName` (a wrapper of `java.lang.String` that enforces the ACL2 constraints on package names) and the symbol's name as a `String`. `Cons` stores the component `Values`. All these fields are private, thus encapsulating the internal representation choices and enabling their localized modification. ACL2 numbers, strings, and symbols have no preset limits, but the underlying Lisp runtime may run out of memory. Their Java representations (e.g. `java.math.BigInteger`) have very large limits, whose exceedance could be regarded as running out of memory. If needed, the Java representations could be changed to overcome the current limits (e.g. by using lists of `java.math.BigIntegers`).

The public classes for ACL2 values and package names provide public static factory methods to build objects of these classes. For example, `Character.make(char)` returns a `Character` with the supplied argument as code, throwing an exception if the argument is above 255. As another example, `Cons.make(Value,Value)` returns a `Cons` with the supplied arguments as components. Some classes provide overloaded variants, e.g. `Integer.make(int)` and `Integer.make(java.math.BigInteger)`. All these classes provide no public Java constructors, thus encapsulating the details of object creation and re-use, which is essentially transparent to external code because these objects are immutable.

The public classes for ACL2 values provide public instance getter methods to unbuild (i.e. extract information from) objects of these classes. For example, `Character.getJavaChar()` returns the code of the character as a `char` that is always below 256. As another example, `Cons.getCar()` and `Cons.getCdr()` return the component `Values` of the `cons` pair. Some classes provide variants, e.g. `Integer.getJavaInt()` (which throws an exception if the integer does not fit in an `int`) and `Integer.getJavaBigInteger()`.

3.2 Terms

ACL2 translates the terms supplied by the user, which may include macros and named constants, into a restricted internal form, in which macros and named constants are expanded [2, `:doc term`]. In the rest of this paper, 'term' means 'translated term', i.e. a term in the restricted internal form.

The set of ACL2 terms consists of (i) variables, which are symbols, (ii) quoted constants, which are lists `(quote` *value*`)` where *value* is a value, and (iii) function applications, which are lists `(`*fn* arg_1 `...` arg_n`)` where *fn* is a function and arg_1, ..., arg_n are zero or more terms. A function *fn* used in a term is (i) a named function, which is a symbol, or (ii) a lambda expression, which is a list `(lambda (`var_1 `...` var_m`)` *body*`)` where var_1, ..., var_m are zero or more symbols and *body* is a term, whose free variables are all among var_1, ..., var_m (i.e. lambda expressions are always closed).

AIJ represents ACL2 terms in a manner similar to ACL2 values, as immutable objects of `Term` and its subclasses in Figure 1; functions are represented as immutable objects of `Function` and its subclasses in Figure 1. The superclasses are abstract, while the subclasses are concrete. All these classes are public.

The information about the represented ACL2 terms is stored in private fields of the non-abstract classes. `Variable` and `NamedFunction` are wrappers of `Symbol`. `Constant` is a wrapper of `Value`.[5] `FunctionApplication` stores a `Function` and an array of zero or more `Terms`. `LambdaExpression` stores an array of zero or more `Variables` and a `Term`.

The non-abstract classes for ACL2 terms (and functions) provide public static factory methods to build objects of these classes, but no public Java constructors, similarly to the classes for ACL2 values.

3.3 Environment

ACL2 terms are evaluated in an environment that includes function definitions, package definitions, etc. AIJ stores information about part of this environment in `Environment` in Figure 1. Since there is just one environment at a time in ACL2, this class has no instances and only static fields and methods.

An ACL2 function definition consists of several pieces of information, of which AIJ only stores (i) the name, which is a symbol, (ii) the parameters, which are zero or more symbols, and (iii) the body, which is a term. `Environment` stores function definitions in a private static field, as a `java.util.Map` from `Symbols` for the functions' names to `LambdaExpressions` for the functions' parameters and bodies. The public static method `Environment.addFunctionDef(Symbol,Symbol[],Term)` adds a function definition to the map.

An ACL2 package definition associates a list of imported symbols to a package name. `Environment` stores package definitions in a private static field, as a `java.util.Map` from `PackageNames` for the packages' names to `java.util.Lists` of `Symbols` for the packages' import lists. The public static method `Environment.addPackageDef(PackageName,List<Symbol>)` adds a package definition to the map. AIJ uses this field to implement the primitive function `pkg-imports`. AIJ also uses information derived from this field to implement the overloaded factory methods `Symbol.make` that build symbols: for instance, `Symbol.make("ACL2","CONS")` returns a `Symbol` with name `"CONS"` and package name `"COMMON-LISP"`, not package name `"ACL2"`, because `"ACL2"` imports `cons` from `"COMMON-LISP"`; the call `Symbol.make("ACL2","CONS")` is the Java equivalent of the ACL2 symbol notation `acl2::cons`.

`Environment` also stores the value of the ACL2 constant `*pkg-witness-name*` in a private static field, as a `java.lang.String`. This field may be set, at most once (otherwise an exception is thrown),

[5]These wrappers place `Symbols` and `Values` into the class hierarchy of `Term` and `Function`, given that Java does not support multiple class inheritance. For instance, `Symbol` could not be both a subclass of `Value` and a subclass of `Term`.

via the public static method `Environment.setPackageWitnessName(java.lang.String)`. AIJ uses this field to implement the primitive function `pkg-witness`.

3.4 Primitive Functions

Since the ACL2 primitive functions have no definitions, AIJ cannot evaluate their calls via their bodies as described in Section 3.5. AIJ implements these functions "natively" in Java, in the package-private class `Primitive` in Figure 1. Each primitive function except `if`, whose calls are evaluated non-strictly as described in Section 3.5, is implemented by a private static method `Primitive.exec`*`Prim`*`(Value)` or `Primitive.exec`*`Prim`*`(Value,Value)` (based on the function's arity), where *`Prim`* is a Java "version" of the function's name; the method returns a `Value`. For instance, `Primitive.execCharCode(Value)` implements `char-code`. The package-private static method `Primitive.call(Symbol,Value[])` evaluates a call of the primitive function named by the `Symbol` argument on the values in the `Value[]` argument, by calling the appropriate `Primitive.exec`*`Prim`* method and returning the result `Value`. `Primitive` has no fields and only the above static methods; no instances of this class are created.

The recognizers `integerp`, `consp`, etc. are implemented to return a `Symbol` for `t` or `nil`, based on whether the argument `Value` is an instance of `Integer`, `Cons`, etc.

The destructors `car`, `numerator`, etc. are implemented to return information from the private fields.

The constructors `complex` and `cons` are implemented via the factory methods `Number.make` and `Cons.make`. The constructor `intern-in-package-of-symbol` is implemented via the factory method `Symbol.make`, and it also calls the getter method `Symbol.getPackageName` on the second argument.

The conversions `char-code` and `code-char` are implemented by passing information from the private field of one class to the factory method of the other class. The conversion `coerce` has a slightly more laborious implementation, which scans or builds a Java representation of an ACL2 list.

The arithmetic operation `unary--` is implemented via package-private instance methods `negate()` in the numeric classes. `Primitive.execUnaryMinus`, when given x as argument, calls x`.negate()`. Dynamic dispatch selects a `negate` method based on the runtime class of x. `Integer.negate` calls `java.math.BigInteger.negate` on its private field and uses the result to return a negated `Integer`. Since $-(a/b) = (-a)/b$, `Ratio.negate` calls `Integer.negate` on the numerator, and uses the result to return a negated `Ratio`. Since $-(a+bi) = (-a)+(-b)i$, `ComplexRational.negate` calls `Rational.negate` on the real and imaginary parts, and uses the returned `Rational`s to return a negated `ComplexRational`; each of these two calls to `Rational.negate` is, in turn, dynamically dispatched to `Integer.negate` or `Ratio.negate`, based on the runtime classes of the real and imaginary parts.

The arithmetic operation `binary-+` is similarly implemented via package-private instance methods `add(Value)` in the numeric classes. But the presence of the second argument leads to a slightly more complicated interplay among the methods. `Primitive.execBinaryPlus`, when given x and y as arguments, calls x`.add(`y`)`, dynamically dispatching based on the runtime class of x. `Integer.add` splits into two cases: if y is an `Integer`, an `Integer` sum is returned using `java.math.BigInteger.add`; otherwise, the roles of x and y are swapped, exploiting the commutativity of addition, by calling y`.add(`x`)`, which dynamically dispatches to a different `add` method. `Ratio.add` performs an analogous split; since $a/b + c/d = (ad+cb)/bd$, this method calls the `multiply` methods, further complicating the method interplay. Since $(a+bi)+(c+di) = (a+c)+(b+d)i$, `ComplexRational.add` calls `Rational.add` on the real and imaginary parts, which are further dynamically dispatched to `Integer.add` or `Ratio.add`.

The arithmetic operations `unary-/` and `binary-*` are implemented analogously to `unary--` and `binary-+`, via `reciprocate()` and `times(Value)` methods in the numeric classes. There is some additional interplay among methods: for instance, since $1/(a+bi) = (a/(a^2+b^2)) - (b/(a^2+b^2))i$, `ComplexRational.reciprocate` calls all the arithmetic methods of `Rational`.

The arithmetic comparison `<` is implemented analogously to `binary-+` and `binary-*`, as part of an implementation of ACL2's total order [2, `:doc lexorder`] via `compareTo(Value)` methods in `Value` and its subclasses, which all implement the `java.lang.Comparable<Value>` interface. In the `compareTo` methods in the numeric classes, when the roles of x and y are swapped in the same way as in the `add` methods described above, the result is negated before being returned, because comparison, unlike addition and multiplication, is not commutative.

The equality `equal` is implemented via methods `equals(java.lang.Object)` in `Value` and its subclasses, which override `java.lang.Object.equals`. These equality methods are implemented in the obvious way.

The implementation of `bad-atom<=` returns a `Symbol` for `nil`, consistently with the raw Lisp code.

The functions `pkg-imports` and `pkg-witness` are implemented as discussed in Section 3.3. They throw an exception if the argument does not name a defined package, matching ACL2's behavior.

All of these implementations do not check guards. They handle `Values` outside the guards according to the applicable ACL2 completion axioms.

3.5 Evaluation

AIJ evaluates ACL2 terms via (i) `eval(java.util.Map<Symbol,Value>)` methods in `Term` and its subclasses, and (ii) `apply(Value[])` methods in `Function` and its subclasses. This evaluation approach is well known [12].

The `eval` methods take maps as arguments that bind values to variable symbols, and return `Value` results. `Constant.eval` returns the constant's value, ignoring the map. `Variable.eval` returns the value bound to the variable, throwing an exception if the variable is unbound. `Application.eval` recursively evaluates the argument terms and then calls `Function.apply` on the function and resulting values. However, if the function represents `if`, `Application.eval` first evaluates just the first argument, and then, based on the result, either the second or third argument, consistently with the non-strictness of `if`.

The `apply` methods take arrays of zero or more `Values` as arguments, and return `Value` results. `LambdaExpression.apply` evaluates the lambda expression's body with a freshly created map that binds the values to the parameters—no old bindings are needed, because lambda expressions are closed. `NamedFunction.apply` calls a public method `Environment.call(Symbol,Value[])` with the name of the function and the argument values. `Environment.call` operates as follows: if the symbol names a primitive function, it is forwarded, with the values, to `Primitive.call`; if the symbol names a function defined in the environment, the lambda expression that defines the function is applied to the values; if the symbol does not name a primitive or defined function, an exception is thrown.

AIJ evaluates ACL2 terms in a purely functional way, without side effects.[6] AIJ does not check the guards of primitive or defined functions. The aforementioned method `Environment.call` calls ACL2 functions on values only, not on (names of) stobjs.

3.6 Usage

AIJ is designed to be used as follows by Java code outside AIJ's package:

[6]Aside from exhausting the available memory, which is, unavoidably, always a possibility.

1. Define all the ACL2 packages of interest by repeatedly calling `Environment.addPackageDef`. For each package, use the factory methods of `PackageName` and `Symbol` to build the name and the imported symbols. Define both the built-in and user-defined packages, in the order in which they appear in the ACL2 history. This order ensures that `Symbol.make` does not throw an exception due to an unknown package.
2. Define all the ACL2 functions of interest by repeatedly calling `Environment.addFunctionDef`. For each function, use the factory methods of the value and term classes to build the name, the parameters, and the body. The functions can be defined in any order, so long as all the packages are defined before the functions (see step above).
3. Call `Environment.setPackageWitnessName` with the appropriate value from the ACL2 constant `*pkg-witness-name*`.
4. Call an ACL2 primitive or defined function as follows:
   (a) Build the name of the ACL2 function to call, as well as zero or more ACL2 values to pass as arguments, via the factory methods of the value classes.
   (b) Call `Environment.call` with the `Symbol` that names the function and the `Value` array of arguments.
   (c) Unbuild the returned `Value` as needed to inspect and use it, using the getter methods of the value classes.
5. Go back to step 4 as many times as needed.

The above protocol explains why AIJ provides a public API for unbuilding ACL2 values but no public API to unbuild the other ACL2 entities (terms etc.). The latter are built entirely by Java code outside AIJ's package, which therefore has no need to unbuild the entities that it builds. Values, instead, may be built by executing ACL2 code that returns them as results: Java code outside AIJ's package may need to unbuild the returned values to inspect and use them.

Besides the structural constraints implicit in the Java classes, and the existence of the referenced packages when building symbols (necessary to resolve imported symbols), AIJ does not enforce any well-formedness constraints when building terms and other entities, e.g. the constraint that the number of arguments in a function call matches the function's arity. However, during evaluation, AIJ makes no well-formedness assumptions and performs the necessary checks, throwing informative exceptions if these checks fail.

4 ATJ: The Code Generator

ATJ is an ACL2 tool that provides an event macro to generate Java code from specified ACL2 functions. The generated Java code provides a public API to (i) build an AIJ representation of the ACL2 functions and other parts of the ACL2 environment and (ii) evaluate calls of the functions on ACL2 values via AIJ. The Java code generated by ATJ automates steps 1, 2, and 3 in Section 3.6 and provides a light wrapper for step 4b, while steps 4a and 4c must be still performed directly via AIJ's API.

ATJ consists of a few thousand lines of ACL2 code (including blank lines, implementation-level documentation, and comments), accompanied by a few hundred lines of user-level documentation in XDOC. The implementation is thoroughly documented in XDOC as well.

4.1 Overview

ATJ generates a single Java file containing a single class, with the following structure:

```
package pname; // if specified by the user
import edu.kestrel.acl2.aij.*; // all the AIJ classes
import ... // a few classes of the Java standard library
public class cname { // 'ACL2' if not specified by the user
    // field to record if the ACL2 environment has been built or not:
    private static boolean initialized = false;
    // one method like this for each known ACL2 package:
    private static void addPackageDef_hex(...) ...
    // one method like this for each specified ACL2 function:
    private static void addFunctionDef_hex1_hex2(...) ...
    // API method to build the ACL2 environment:
    public static void initialize() ...
    // API method to evaluate ACL2 function calls:
    public static Value call(Symbol function, Value[] arguments) ...
}
```

The file has the same name as the class; it is (over)written in the current working directory, unless the user specifies a directory. ATJ directly generates Java concrete syntax, via formatted printing to the ACL2 output channel associated to the file, without going through a Java abstract syntax and pretty printer.

4.2 Value and Term Building

As part of building an AIJ representation of the ACL2 environment, the Java code generated by ATJ builds AIJ representations of ACL2 values and terms: function definitions include terms as bodies, and constant terms include values. It does so via the factory methods discussed in Sections 3.1 and 3.2.

In principle, ATJ could turn each ACL2 value or term into a single Java expression with an “isomorphic” structure. For example, the ACL2 value `((10 . #\A) . "x")` could be built as follows:

```
Cons.make(Cons.make(Integer.make(10),
                    Character.make(65)),
          String.make("x"))
```

However, values and terms of even modest size (e.g. function bodies) would lead to large expressions, which are not common in Java. Thus, ATJ breaks them down into sub-expressions assigned to local variables. For instance, the example value above is built as follows:

```
// statements:
Value value1 = Integer.make(10);
Value value2 = Character.make(65);
Value value3 = Cons.make(value1, value2);
Value value4 = String.make("x");
// expression:
Cons.make(value3, value4)
```

In general, ATJ turns each ACL2 value or term into (i) zero or more Java statements that incrementally build parts of it and (ii) one Java expression that builds the whole of it from the parts. ATJ does so recursively: the expression for a sub-value or sub-term is assigned to a new local variable that is used in the expression for the containing super-value or super-term. The top-level expressions are used as explained in Sections 4.3 and 4.4.

To generate new local variable names, ATJ keeps track of three numeric indices (for values, terms, and lambda expressions—recall that the latter are mutually recursive with terms) as it recursively traverses values and terms. The appropriate index is appended to '`value`', '`term`', or '`lambda`' and then incremented.

4.3 Package Definition Building

The Java code generated by ATJ builds an AIJ definition of each ACL2 package known when ATJ is invoked. The names of the known packages are the keys of the alist returned by the built-in function `known-package-alist`, in reverse chronological order.

The AIJ definition of each of these packages is built by a method `addPackageDef_`*`hex`* (see Section 4.1), where *`hex`* is an even-length sequence of hexadecimal digits for the ASCII codes of the characters that form the package name. For instance, the definition of the `"ACL2"` package is built by `addPackageDef_41434C32`. This simple naming scheme ensures that the generated method names are distinct and valid, since ACL2 package names allow characters disallowed by Java method names.

Each `addPackageDef_`*`hex`* method builds a Java list of all the symbols imported by the package, which ATJ obtains via `pkg-imports`. Then the method calls `Environment.addPackageDef` with the `PackageName` and the list of `Symbols`.

4.4 Function Definition Building

The Java code generated by ATJ builds an AIJ definition of each (non-primitive) ACL2 function specified via one or more function symbols fn_1, ..., fn_p supplied to ATJ. Each fn_i implicitly specifies not only fn_i itself, but also all the functions called directly or indirectly by fn_i, ensuring the "closure" of the generated Java code under ACL2 calls.

ATJ uses a worklist algorithm, initialized with (fn_1 ... fn_p), to calculate a list of their closure under calls. Each iteration removes the first function *fn* from the worklist, adds it to the result list, and extends the worklist with all the functions directly called by *fn* that are not already in the result list. Here 'directly called by' means 'occurring in the `unnormalized-body` property of'; occurrences in the guard of *fn* do not count, because ATJ, like AIJ, ignores guards. If *fn* has no `unnormalized-body` property, it must be primitive, otherwise ATJ stops with an error—this happens if *fn* is a constrained, not defined, function. If *fn* is in `logic-fns-with-raw-code` or `program-fns-with-raw-code` (see Section 2), it must be in a whitelist of functions that are known to have no side effects;[7] If *fn* has input or output stobjs, ATJ stops with an error—this may only happen if *fn* is not primitive.

The AIJ definition of each of these functions is built by a method `addFunctionDef_`*`hex1`*`_`*`hex2`* (see Section 4.1), where *`hex1`* and *`hex2`* are even-length sequences of hexadecimal digits for the ASCII codes of the package and symbol names of the function symbol. For instance, the definition of the `len` function is built by `addFunctionDef_41434C32_4C454E`. This simple naming scheme ensures that the generated method names are distinct and valid, since ACL2 package and symbol names allow characters disallowed by Java method names.

Each `addFunctionDef_`*`hex1`*`_`*`hex2`* method first builds a `Term` from the `unnormalized-body` property of the function, as explained in Section 4.2. Then the top-level Java expression, along with a `Symbol` for the function name and with a `Variable` array for the function parameters, is passed to `Environment.addFunctionDef`.

[7]This whitelist is currently a subset of `logic-fns-with-raw-code`. It consists of functions whose raw Lisp code makes them run faster but is otherwise functionally equivalent to the ACL2 definitions.

4.5 Environment Building

The `initialize` method generated by ATJ (see Section 4.1) calls all the `addPackageDef_`*hex* and `addFunctionDef_`*hex1*`_`*hex2* methods described in Sections 4.3 and 4.4. The method also calls `Environment.setPackageWitnessName` with an argument derived from `*pkg-witness-name*`.

The package definition methods are called in the same order in which the corresponding packages are defined, which is the reverse order of the alist returned by `known-package-alist`.

This ensures the success of the calls of `Symbol.make` that build the elements of a package's import list. For instance, if `"P"` imports `q::sym`, then `"Q"` must be already defined when `"P"` is being defined. That is, `addPackageDef_51` must have already been called when `addPackageDef_50` calls `Symbol.make("Q","SYM")` as part of building `"P"`'s import list, which is needed to define `"P"`; otherwise, `Symbol.make("Q","SYM")` would throw an exception due to `"Q"` being still undefined.

The function definition methods are called after the package definition methods, again to ensure the success of the `Symbol.make` calls. The relative order of the function definitions is unimportant; the result list returned by ATJ's worklist algorithm (see Section 4.4) is in no particular order.

The `initialize` method may be called once by external code: the method throws an exception unless the `initialized` field (see Section 4.1) is `false`, and sets the field to `true` just before returning.

4.6 Call Forwarding

The *`cname`*`.call` method generated by ATJ (see Section 4.1) forwards the function name and the argument values to `Environment.call`, after ensuring that the `initialized` field is `true`, i.e. that the ACL2 environment has been built. It throws an exception if `initialized` is still `false`.

5 Preliminary Tests and Optimizations

The initial version of AIJ was deliberately written in a very simple way, without regard to performance, as a sort of "executable specification" in Java. The reasons were to increase assurance by reducing the chance of errors, facilitate the potential verification of the code, avoid premature optimizations, and observe the impact of gradually introduced optimizations.

Performance has been tested mainly on three example programs. The first is an ACL2 function that computes factorial non-tail-recursively. The second is an ACL2 function that computes Fibonacci non-tail-recursively. The third is a slightly modified version of the verified ABNF grammar parser [5] from the ACL2 Community Books [2, `:doc abnf::grammar-parser`]: the parser, in the `:logic` part of `mbe`, calls `nat-list-fix` on its input list of natural numbers just before reading each natural number, which makes execution "in the logic" (which is how AIJ executes) unduly slow; for testing AIJ more realistically, the parser was tweaked to avoid these calls of `nat-list-fix`. The tweaked parser is about 2,000 lines (including blank lines), including theorems to prove its termination so that it is in logic mode, and including return type theorems to prove its guards. The parser not only recognizes ABNF grammars, but also returns parse trees.[8]

Unsurprisingly, the initial version of AIJ was quite slow. A re-examination of the code from a performance perspective readily revealed several easy optimization opportunities, which were carried out and are part of the current version of AIJ. These are the main ones, in order:

[8]Initially, tests were conducted on a simplified version of the parser that only recognized ABNF grammars, because ATJ did not support `mbe`, which is used in the construction of parse trees (defined via fixtypes [2, `:doc fty`]). After extending ATJ to support `mbe`, testing was switched to the more realistic version of the parser that also returns parse trees.

1. The `Character` array representation of `Strings` was replaced with `java.lang.String`.
2. `Symbols` frequently used during evaluation, such as the ones for `t`, `nil`, and the names of the primitive functions, were cached as constants instead of being built repeatedly.
3. `Characters` were interned, as follows. `Character` objects for all the 256 codes were pre-created and stored into an array, in the order of their codes. The factory method `Character.make(char)` indexes the array with the input code and returns the corresponding object. Since this ensures that there is just one object for each character code, `Character.equals` uses pointer equality (i.e. ==) and the default fast `java.lang.Object.hashCode` is inherited.
4. `PackageNames`, `Strings`, and `Symbols` were interned, similarly to `Characters`, as follows. Since there is a potentially infinite number of them, they are are created on demand. For each of these three classes, all the objects created thus far are stored as values of a `java.util.Map`, whose keys are `java.lang.Strings` for `PackageName` and `String`, and `PackageNames` paired with `Strings` for `Symbol`—the pairing is realized via nested maps. Each factory method first consults the appropriate map, either returning the existing object, or creating a new one that is added to the map. Similarly to `Character`'s interning, the interning of these classes enables the use of pointer equality in the equality methods and the inheritance of the default fast hash code method.

Thanks to AIJ's object-oriented encapsulation, all these optimizations were easy and localized. These optimizations did not involve ATJ, because the code generated by ATJ is essentially used just to initialize the ACL2 environment (see Section 4.5), which happens quickly for the factorial and Fibonacci functions and for the ABNF parser.

Based on a few time measurements on the ABNF parser and a few other artificial programs, the above optimizations reduced execution time, very roughly, by the following factors, one after the other: 2 for optimization #1, 5 for optimization #2, and 2 for optimizations #3 and #4—all combined, 20.

Tables 1, 2, and 3 report more systematic time measurements for the factorial function, Fibonacci function, and ABNF parser. Each row corresponds to an input of the program: natural numbers for the factorial and Fibonacci functions; ABNF grammars (all from Internet standards, including ABNF itself) for the ABNF parser. The 'ACL2' columns are for execution in ACL2, with guard checking ('g.c.') set to `t`, i.e. typical execution, and `:none`, i.e. execution "in the logic"; the latter matches AIJ's execution. The 'AIJ' column is for execution with AIJ's current version. Each cell contains minimum, average, and maximum real times from 10 runs, in seconds rounded to the millisecond. The ACL2 times were measured as the difference between the results of `read-run-time` just before and just after the call of the factorial function, Fibonacci function, or top-level ABNF parsing function. The Java times were measured as the difference between the results of `java.lang.System.currentTimeMillis()` just before and just after the call of *`cname`*`.call` on the corresponding ACL2 function. Given the AIJ evaluator's recursive implementation, a larger stack size than the default must be passed to the JVM (1 GB for these time measurements) to avoid a stack overflow.

The times in Table 1 are all roughly comparable for each input, with ACL2 faster on smaller inputs and AIJ faster on larger inputs: presumably, most of the time is spent multiplying large numbers, which all happens in `java.math.BigInteger` in the Java code and in Lisp's bignum implementation in the ACL2 code, dwarfing the contributions of ACL2 and AIJ proper, especially for the larger inputs.[9] The times in Table 2 differ: looking at the averages, AIJ is about 17–30 times slower than ACL2 with guard checking `:none`, which is about 8–10 times slower than ACL2 with guard checking `t`. The times in Table 3 differ as well: looking at the averages, AIJ is about 19–22 times slower than ACL2 with guard checking `:none`, which is about 16–87 times slower than ACL2 with guard checking `t`; nonetheless,

[9]Even the initial, unoptimized version of AIJ took comparable times.

the absolute times suggest that the Java code of the parser is usable.[10] Performance needs vary: AIJ's current speed may be adequate to some applications, such as security-critical interactive applications like cryptocurrency wallets. Furthermore, as discussed in Section 6, there are more opportunities to optimize AIJ.

Input	**ACL2 [g.c. `t`]**			**ACL2 [g.c. `:none`]**			**AIJ**		
	min	**avg**	**max**	**min**	**avg**	**max**	**min**	**avg**	**max**
1,000	0.000	0.000	0.001	0.000	0.000	0.001	0.003	0.005	0.012
5,000	0.007	0.011	0.022	0.007	0.009	0.011	0.009	0.029	0.059
10,000	0.031	0.035	0.038	0.032	0.034	0.040	0.026	0.035	0.068
50,000	1.324	1.355	1.432	1.319	1.328	1.337	0.589	0.687	1.044
100,000	6.280	6.385	6.604	6.279	6.291	6.307	2.340	2.547	2.705

Table 1: Time measurements for the factorial function.

Input	**ACL2 [g.c. `t`]**			**ACL2 [g.c. `:none`]**			**AIJ**		
	min	**avg**	**max**	**min**	**avg**	**max**	**min**	**avg**	**max**
10	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.001	0.004
20	0.000	0.000	0.000	0.001	0.001	0.001	0.019	0.030	0.053
30	0.007	0.008	0.009	0.061	0.063	0.079	1.043	1.094	1.210
40	0.727	0.734	0.750	7.144	7.205	7.355	126.167	127.149	129.959

Table 2: Time measurements for the Fibonacci function.

Input	**ACL2 [g.c. `t`]**			**ACL2 [g.c. `:none`]**			**AIJ**		
	min	**avg**	**max**	**min**	**avg**	**max**	**min**	**avg**	**max**
ABNF grammar	0.004	0.007	0.014	0.109	0.113	0.117	2.391	2.478	3.076
JSON grammar	0.001	0.002	0.006	0.044	0.049	0.053	1.011	1.023	1.031
URI grammar	0.002	0.003	0.006	0.100	0.105	0.112	2.218	2.233	2.251
HTTP grammar	0.002	0.004	0.010	0.167	0.175	0.189	3.577	3.597	3.626
IMF grammar	0.009	0.014	0.021	1.028	1.079	1.414	21.173	21.522	21.741
SMTP grammar	0.007	0.010	0.019	0.398	0.404	0.411	8.648	8.733	8.896
IMAP grammar	0.020	0.026	0.030	2.198	2.267	2.481	43.083	43.490	43.805

Table 3: Time measurements for the ABNF parser.

All the time measurements were taken on a MacBook Pro (15-inch, 2017) with 3.1 GHz Intel Core i7 and 16 GB 2133 MHz LPDDR3, running macOS High Sierra Version 10.13.6. The ACL2 times were measured with commit 852ee0aca96deac2b3c062ee03f458acca668f6e from GitHub running on 64-bit Clozure Common Lisp Version 1.11.5. The Java times were measured with the version of AIJ in the same commit from GitHub as above, running on Oracle's 64-bit Java 10 2018-03-20, Java SE Runtime Environment 18.3 (build 10+46). Just before taking the measurements, the machine was rebooted and only the necessary applications were started.

The performance of ATJ does not affect the performance of the Java code. ATJ runs in 1–2 seconds on each of the factorial function, Fibonacci function, and ABNF parser, including the time to write the Java files; this was measured by wrapping the calls of ATJ with `time$`.

[10] As another data point, the simplified parser mentioned in Footnote 8 was about 4–7 times faster than the current parser.

6 Future Work

Evaluating non-executable functions (i.e. non-primitive and without an `unnormalized-body` property), by throwing an exception that mirrors the error that ACL2 yields, is easy but not necessarily useful.[11] A planned extension is to support guards and evaluation with different guard-checking settings, in the same way as ACL2. Support for functions with side effects will be added one at a time, by writing native Java implementations (as done for the primitive functions) that suitably mirror the ACL2 side effects in Java; for instance, hard errors could be implemented as exceptions. User-defined stobjs could be supported by storing their contents in Java fields that are destructively updated; since `state` is "linked" to external entities (e.g. the file system), support for this built-in stobj will involve the use of the Java API of those entities. Supporting stobjs also involves extending AIJ's public API to call ACL2 functions on stobj names, besides values. Direct support for calling macros directly, and for supplying named constants to function calls, are also candidate extensions.

The generated method *`cname`*`.call` described in Section 4.6 does not provide much beyond calling `Environment.call` directly, but is suggestive of additional functionality. For example, future versions of *`cname`* could provide a public method for each top-level target function fn_i supplied to ATJ, with no parameter for the function name, and with as many `Value` parameters as the function's arity instead of a single `Value` array parameter. As another example, *`cname`* could provide additional public methods to call each fn_i on objects of more specific types (e.g. `Integer` instead of `Value`), based on the guards. The names of these methods should be derived from the names of the corresponding functions, according to safe but more readable schemes than the one described in Section 4.4—in fact, a more readable scheme should be used for the methods described in Section 4.3 and Section 4.4 as well.

A reviewer suggested to make the fields and methods of `Environment` non-static and have multiple instances of this class at once. This is worth exploring.

There are more optimization opportunities beyond the ones already carried out and described in Section 5. For example, now each variable evaluation looks up the variable symbol in the hash map that stores the current binding of values to variables. As another example, now each function call first looks up the function symbol in the hash map that stores the function definitions in the environment, and then, if no definition is found, it compares the function symbol with all the primitive function symbols until a match is found. Replacing or enhancing AIJ's representation of variable and function symbols with numeric indices should make all these accesses much faster.[12] As a third example, AIJ's evaluator could be re-implemented as a loop with an explicit stack, instead of a recursion. As a fourth example, many built-in ACL2 functions could be implemented natively in Java (as done for the primitive functions), instead of being interpreted.

A reviewer suggested to implement `hons` [2, `:doc hons-and-memoization`] in AIJ, and use it instead of `cons`. This amounts to interning all the Java objects that represent ACL2 values (not just `Character`s, `String`s, and `Symbol`s—see Section 5), enabling fast equality tests and hash code computations, which could increase performance in some applications. Perhaps future versions of AIJ and ATJ could provide options to use `hons` vs. `cons`.

The eventual path to fast execution is to avoid the interpretation overhead, by having ATJ turn ACL2 functions into shallowly embedded Java representations, as is customary in code generators.[13] The shal-

[11] If support for evaluating non-executable function is added, ATJ should still include an option to signal an error when the worklist algorithm reaches a non-executable function.

[12] A preliminary experiment with just variable indices seems to reduce execution times roughly by 2.

[13] Besides more conventional translation approaches, a more speculative idea is to generate the shallowly embedded representations by partially evaluating [10] the AIJ interpreter on the deeply embedded representations generated by ATJ.

low embedding will consist of Java methods that implement the ACL2 functions, with suitably matching signatures. AIJ's representation of and operations on ACL2 values will still be used, but AIJ's representation and evaluator of ACL2 terms will not. Under certain conditions, it should be possible to generate variants of these Java methods that use more efficient representations and operations, e.g. the Java `int` values and integer operations when there is provably no wrap-around, in particular leveraging ACL2's `the` forms and the associated guard verification, which similarly help the Lisp compiler. Given that generating these shallowly embedded representations is inevitably more complicated and thus error-prone, the slower but safer interpreted evaluation could be still available as an option, at least in the absence of verification.

The Java code generated by ATJ can be called by external Java code, but not vice versa. Allowing the other call direction may involve suitable ACL2 stubs that correspond to the external code to be called.

The implementation of ATJ could be simplified by directly generating a Java abstract syntax and using a separable pretty printer to write abstract syntax to the file.

More ambitious projects are to (i) verify the correctness of AIJ's evaluator and primitive function implementations, and (ii) extend ATJ to generate a proof of correctness of the generated Java code, like a verifying compiler. Optimizing AIJ and generating shallowly embedded representations in ATJ make these verification tasks harder; an idea worth exploring is to perform a compositional verification of optimized Java code against unoptimized Java code and of the latter against ACL2 code.

The approach to generate Java code described in this paper, including the envisioned extensions described in this section, could be used to generate code in other programming languages. In particular, the UML class diagram in Figure 1 could be used for other object-oriented programming languages.

7 Related Work

The author is not aware of any Java or other code generator for ACL2.

Several theorem provers (Isabelle, PVS, Coq, etc.) include facilities to generate code in various programming languages (Standard ML, Ocaml, Haskell, Scala, C, Scheme, etc.) [8, 13, 7]. These code generators use shallow-embedding approaches, very different from ATJ and AIJ's deep-embedding approach. These code generators may be more relevant to future versions of ATJ and AIJ that use a shallow-embedding approach (see Section 6). However, the ACL2 language is quite different from the languages of those provers: first-order vs. higher-order, untyped with a fixed universe of (evaluation) values vs. typed with user-definable types, extra-logical guards vs. types that are part of the logic, and so on. Thus, only some of the ideas from those provers' code generators may be relevant to ACL2.

As discussed in Section 1, there are other ways for ACL2 code to interoperate with code in other programming languages, without the need for generating code in those programming languages from ACL2. However, this obviated need should be balanced against the issues with these approaches discussed in Section 1; different approaches may be best suited to different applications.

Acknowledgements

This work was supported by DARPA under Contract No. FA8750-15-C-0007. Thanks to Matt Kaufmann for useful discussions about the ACL2 evaluation semantics. Thanks to the anonymous reviewers for valuable suggestions.

References

[1] *Armed Bear Common Lisp (ABCL).* `https://abcl.org`.

[2] *ACL2 Theorem Prover and Community Books: User Manual.* `http://www.cs.utexas.edu/~moore/acl2/manuals/current/manual`.

[3] *APT (Automated Program Transformations).* `http://www.kestrel.edu/home/projects/apt`.

[4] *CFFI: The Common Foreign Function Interface.* `https://common-lisp.net/project/cffi`.

[5] Alessandro Coglio (2018): *A Formalization of the ABNF Notation and a Verified Parser of ABNF Grammars.* In: *Proc. 10th Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE).* To appear in Springer LNCS.

[6] Alessandro Coglio, Matt Kaufmann & Eric Smith (2017): *A Versatile, Sound Tool for Simplifying Definitions.* In: *Proc. 14th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2017)*, pp. 61–77, doi:10.4204/EPTCS.249.5.

[7] *Coq 8.8.1 Reference Manual.* `https://coq.inria.fr`.

[8] Florian Haftmann with contributions from Lukas Bulwahn (2017): *Code generation from Isabelle/HOL theories.* `https://isabelle.in.tum.de`. Tutorial distributed with Isabelle/HOL.

[9] *Java Native Interface JNI Specification.* `https://docs.oracle.com/javase/10/docs/specs/jni`.

[10] Neil D. Jones, Carsten K. Gomard & Peter Sestoft (1999): *Partial Evaluation and Automatic Program Generation.* Prentice Hall. `http://www.itu.dk/people/sestoft/pebook`.

[11] Matt Kaufmann & J Strother Moore (1998): *A Precise Description of the ACL2 Logic.* Technical Report, Department of Computer Sciences, University of Texas at Austin. `http://www.cs.utexas.edu/users/moore/publications/km97a.pdf`.

[12] John McCarthy (1960): *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I.* *Communications of the ACM* 3(4), pp. 184–195, doi:10.1145/367177.367199.

[13] Nararajan Shankar (2017): *A Brief Introduction to the PVS2C Code Generator.* In: *Proc. Workshop on Automated Formal Methods (AFM'17).*

[14] Guy L. Steele (1990): *Common Lisp the Language.* Digital Press. `https://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html`.

Formalising Filesystems in the ACL2 Theorem Prover: an Application to FAT32

Mihir Parang Mehta
Department of Computer Science
University of Texas at Austin
Austin, TX, USA
mihir@cs.utexas.edu

In this work, we present an approach towards constructing executable specifications of existing filesystems and verifying their functional properties in a theorem proving environment. We detail an application of this approach to the FAT32 filesystem.

We also detail the methodology used to build up this type of executable specification through a series of models which incrementally add features of the target filesystem. This methodology has the benefit of allowing the verification effort to start from simple models which encapsulate features common to many filesystems and which are thus suitable for reuse.

1 Introduction and overview

Filesystems are ubiquitous in computing, providing application programs a means to store data persistently, address data by names instead of numeric indices, and communicate with other programs. Thus, the vast majority of application programs directly or indirectly rely upon filesystems, which makes filesystem verification critically important. Here, we present a formalisation effort in ACL2 for the FAT32 filesystem, and a proof of the read-over-write properties for FAT32 system calls. By starting with a high-level abstract model and adding more filesystem features in successive models, we are able to manage the complexity of this proof, which has not, to our knowledge, been previously attempted. Thus, this paper contributes an implementation of several Unix-like system calls for FAT32, formally verified against an abstract specification and tested for binary compatibility by means of co-simulation.

In the rest of this paper, we describe these filesystem models and the properties proved, with examples; we proceed to a high-level explanation of these proofs and the co-simulation infrastructure; and further we offer some insights about the low-level issues encountered while working out the proofs.

2 Related work

Filesystem verification research has largely followed a pattern of synthesising a new filesystem based on a specification chosen for its ease in proving properties of interest, rather than faithfulness to an existing filesystem. Our work, in contrast, follows the FAT32 specification closely. In spirit, our work is closer to previous work which uses interactive theorem provers and explores deep functional properties than to efforts which use non-interactive theorem provers such as Z3 to produce fully automated proofs of simpler properties.

2.1 Interactive theorem provers

An early effort in the filesystem verification domain was by Bevier and Cohen [4], who specified the Synergy filesystem and created an executable model of the same in ACL2 [14], down to the level of processes and file descriptors. They certified their model to preserve well-formedness of their data structures through their various file operations; however, they did not attempt to prove read-over-write properties or crash consistency. Later, Klein et al. with the SeL4 project [23] used Isabelle/HOL [28] to verify a microkernel; while their microkernel design excluded file operations in order to keep their trusted computing base small, it did serve as a precursor to their more recent COGENT project [2]. Here the authors built a verifying compiler to translate a filesystem specification in their domain-specific language to C-language code, accompanied by a proof of the correctness of this translation. Elsewhere, the SibylFS project [29], again using Isabelle/HOL, provided an executable specification for filesystems at a level of abstraction that could function across multiple operating systems including OSX and Unix. The Coq prover [3] has been used to aid the development of FSCQ [6], a state-of-the art filesystem built to have high performance and formally verified crash consistency properties.

2.2 Non-interactive theorem provers

Non-interactive theorem provers such as Z3 [8] have also been used to analyse filesystem models. Hyperkernel [27] is a recent effort which simplifies the xv6 [7] microkernel until the point where Z3 can verify its various properties with its SMT solving techniques. However, towards this end, all system calls in Hyperkernel are replaced with analogs which can terminate in constant time; while this approach is theoretically sound, it increases the chances of discrepancies between the model and the implementation which may diminish the utility of the proofs or even render them moot. A stronger effort in the same domain is Yggdrasil [31], which focusses on verifying filesystems with the use of Z3. While the authors have made substantial progress in terms of the number of filesystem calls they support and the crash consistency guarantees they provide, they are subject to the limits of SMT solving which prevent them from modelling filesystem features such as extents, which are essential to FAT32 and many other filesystems.

3 Program architecture

Modern filesystems, in response to the evolution of end users' needs over time, have developed a substantial amount of complexity in order to serve file operations in a fast and reliable manner. In order to address this complexity in a principled way, we choose to build our filesystem models incrementally, adding filesystem features in each new model and proving equivalence with earlier, simpler models.

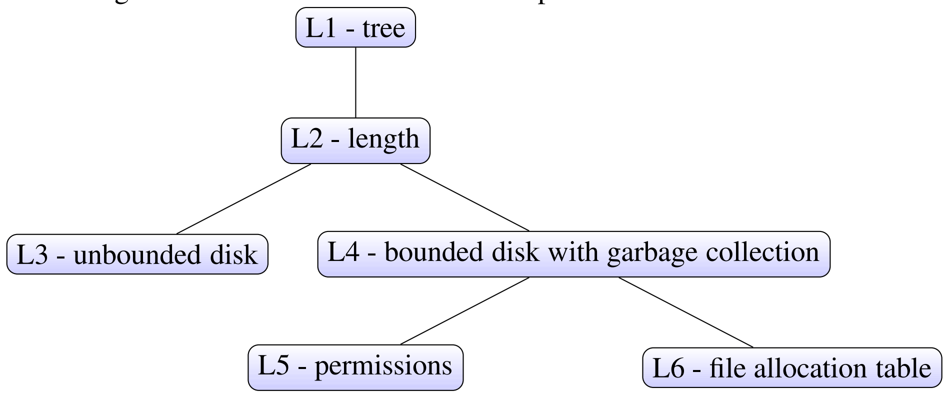
We have two concrete models for the FAT32 filesystem - `M2`, which is a faithful representation of a FAT32 disk image in the form of a stobj [5], and `M1`, which represents the state of the filesystem as a directory tree. This allows us to address the practical details of updating a disk image in `M2`, which benefits from the efficient array operations ACL2 provides for stobjs, and abstract them away in `M1` for easier reasoning without the syntactic constraints imposed on stobj arrays.

These concrete filesystem models are based upon abstract models `L1` through `L6`. Incremental construction allows us to reuse read-over-write proofs for simpler models in more complex models. In the case of models `L4` and `L6`, we are able to show a refinement relationship without stuttering [1]; however, for the other models we are able to reuse proofs without proving a formal refinement relation. These reuse relationships are summarised in figure 1. Much of the code and proof infrastructure is also

Table 1: Abstract models and their features

`L1`	The filesystem is represented as a tree, with leaf nodes for regular files and non-leaf nodes for directories. The contents of regular files are represented as strings stored in the nodes of the tree; the storage available for these is unbounded.
`L2`	A single element of metadata, *length*, is stored within each regular file.
`L3`	The contents of regular files are divided into blocks of fixed size. These blocks are stored in an external "disk" data structure; the storage for these blocks remains unbounded.
`L4`	The storage available for blocks is now bounded. An allocation vector data structure is introduced to help allocate and garbage-collect blocks.
`L5`	Additional metadata for file ownership and access permissions is stored within each regular file.
`L6`	The allocation vector is replaced by a file allocation table, matching the official FAT specification.

Figure 1: Refinement/reuse relationships between abstract models


L1 - tree
L2 - length
L3 - unbounded disk
L4 - bounded disk with garbage collection
L5 - permissions
L6 - file allocation table

shared between the abstract models and the concrete models by design. Details of the filesystem features introduced in the abstract models can be seen in table 1.

A design choice that arises in this work pertains to the level of abstraction: how operating-system specific do we want to be in our model? Choosing, for instance, to make our filesystem operations conform to the `file_operations` interface [30] provided by the Linux kernel for its filesystem modules would make our work less general, but avert us from having to recreate some of the filesystem infrastructure provided by the kernel. We, however, choose to implement a subset of the POSIX filesystem application programming interface, in order to enable us to easily compare the results of running filesystem operations on `M2` and the Linux kernel's implementation of FAT32, which in turn allows us to test our implementation's correctness through co-simulation in addition to theorem proving. As a trade-off for this choice, we are required to implement process tables and file tables, which we do through a straightforward approach similar to that used in Synergy [4].

At the present moment, we have implemented the POSIX system calls `lstat` [17], `open` [20], `pread` [21], `pwrite` [22], `close` [15], `mkdir` [18] and `mknod` [19]. Wherever `errno` [16] is to be set by a system call, we abide by the Linux convention.

4 The FAT32 filesystem

FAT32 was initially developed at Microsoft in order to address the capacity constraints of the DOS filesystem. Microsoft's specification for FAT32 [25], which we follow closely in our work, details the layout of data and metadata in a valid FAT32 disk image.

In FAT32 all files, including regular files and directory files, are divided into *clusters* (sometimes called *extents*) of a fixed size. The size of a cluster, along with other volume-level metadata, is stored in a *reserved area* at the beginning of the volume, and the clusters themselves are stored in a *data region* at the end of the volume. Between these two on-disk data structures, the volume has one or more redundant copies of the *file allocation table*.

The cluster size must be an integer multiple of the sector size, which in turn must be at least 512 bytes; these and other constraints on the fields of the reserved area are detailed in the FAT32 specification.

Directory files are for the most part treated the same way as regular files by the filesystem, but they differ in a metadata attribute, which indicates that the contents of directory files should be treated as sequences of directory entries. Each such directory entry is 32 bytes and contains metadata including name, size, first cluster index, and access times for the corresponding file.

The file allocation table is a table with one entry for each cluster in the data region; it contains a number of linked lists (*clusterchains*). It maps each cluster index used by a file to either the next cluster index for that file or a special end-of-clusterchain value. [1] This allows the contents of a file to be reconstructed by reading just the first cluster index from the corresponding directory entry, reconstructing the clusterchain using the table, and then looking up the contents of these clusters in the data region. Unused clusters are mapped to 0 in the table; this fact is used for counting and allocating free clusters.

We illustrate the file allocation table and data layout for a small example directory tree in figure 2. Here, `/tmp` is a subdirectory of the root directory (`/`). For the purposes of illustration, all regular files and directories in this example are assumed to span one cluster except for `/vmlinuz` which spans two clusters (3 and 4), and *EOC* refers to an "end of clusterchain" value. Also, as shown in the figure, the specification requires the first two entries in the file allocation table (0 and 1) to be considered reserved, and thus unavailable for clusterchains.

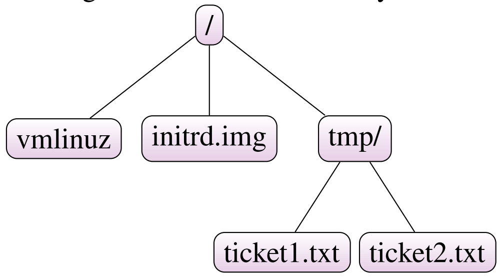
5 Proof methodology

Broadly, we characterise the filesystem operations we offer as either *write* operations, which do modify the filesystem, or *read* operations, which do not. In each model, we have been able to prove *read-over-write* properties which show that write operations have their effects made available immediately for reads at the same location, and also that they do not affect reads at other locations.

The first read-over-write theorem states that immediately following a write of some text at some location, a read of the same length at the same location yields the same text. The second read-over-write theorem states that after a write of some text at some location, a read at any other location returns exactly what it would have returned before the write. As an example, listings for the `L1` versions of these theorems follow. Note, the hypothesis `(stringp (l1-stat path fs))` stipulates that a regular file should exist at `path`, where `l1-stat` is a function for traversing a directory tree to look up a regular file or a directory at a given path (represented as a list of symbols, one for each subdirectory/file name).

[1] There is a range of end-of-clusterchain values in the specification, not just one. We support all values in the range.

Figure 2: A FAT32 directory tree



FAT index	FAT entry
0 (reserved)	
1 (reserved)	
2	*EOC*
3	4
4	*EOC*
5	*EOC*
6	*EOC*
7	*EOC*
8	*EOC*
9	0
⋮	⋮

	Directory entry in /
0	“vmlinuz”, 3
32	“initrd.img”, 5
64	“tmp”, 6
⋮	⋮

	Directory entry in /tmp/
0	“ticket1”, 7
32	“ticket2”, 8
⋮	⋮

```
(defthm l1-read-after-write-1
  (implies (and (l1-fs-p fs)
                (stringp text)
                (symbol-listp path)
                (natp start)
                (equal n (length text))
                (stringp (l1-stat path fs)))
           (equal (l1-rdchs path (l1-wrchs path fs start text) start n) text)))

(defthm l1-read-after-write-2
  (implies (and (l1-fs-p fs)
                (stringp text2)
                (symbol-listp path1)
                (symbol-listp path2)
                (not (equal path1 path2))
                (natp start1)
                (natp start2)
                (natp n1)
                (stringp (l1-stat path1 fs)))
           (equal (l1-rdchs path1 (l1-wrchs path2 fs start2 text2) start1 n1)
                  (l1-rdchs path1 fs start1 n1))))
```

By composing these properties, we can reason about executions involving multiple reads and writes, as illustrated in the following throwaway proof.

```
(thm
 (implies (and (l1-fs-p fs)
               (stringp text1)
               (stringp text2)
               (symbol-listp path1)
               (symbol-listp path2)
               (not (equal path1 path2))
               (natp start1)
               (natp start2)
               (stringp (l1-stat path1 fs))
               (equal n1 (length text1)))
          (equal (l1-rdchs path1
                           (l1-wrchs path2 (l1-wrchs path1 fs start1 text1)
                                     start2 text2)
                           start1 n1)
                 text1)))
```

In `L1`, our simplest model, the read-over-write properties are proven from scratch. In each subsequent model, the read-over-write properties are proven as corollaries of equivalence proofs which establish the correctness of read and write operations in the respective model with respect to a previous model. A representation of such an equivalence proof can be seen in figures 3, 4 and 5, which respectively show the equivalence proof for `l2-wrchs`, the equivalence proof for `l2-rdchs` and the composition of these to obtain the first read-over-write theorem for model `L2`.

Figure 3: l2-wrchs-correctness-1

$$\begin{tikzcd}
l2 \arrow[r, "\text{write}"'] \arrow[d, "\text{l2-to-l1-fs}"'] & l2' \arrow[d, "\text{l2-to-l1-fs}"'] \\
l1 \arrow[r, "\text{write}"'] & l1'
\end{tikzcd}$$

Figure 4: l2-rdchs-correctness-1

$$\begin{tikzcd}
l2 \arrow[r, "\text{read}"'] \arrow[d, "\text{l2-to-l1-fs}"'] & text \\
l1 \arrow[ur, "\text{read}"'] &
\end{tikzcd}$$

6 Proof and implementation details

We have come to rely on certain principles for the development of each new model and its corresponding proofs. We summarise these below.

6.1 Invariants

As the abstract models grow more complex, with the addition of more auxiliary data the “sanity” criteria for filesystem instances become more complex. For instance, in `L4`, the predicate `l4-fs-p` is defined to be the same as `l3-fs-p`, which recursively defines the shape of a valid directory tree. However, we choose to require two more properties for a “sane” filesystem.

1. Each disk index assigned to a regular file should be marked as *used* in the allocation vector - this is essential to prevent filesystem errors.
2. Each disk index assigned to a regular file should be distinct from all other disk indices assigned to files - this does not hold true, for example, in filesystems with hardlinks. FAT32 lacks hardlinks, and we can use this fact to make our proofs easier.

These properties are invariants to be maintained across write operations; while not all of them are strictly necessary for a filesystem instance to be valid, they do simplify the verification of read-over-write properties by helping us ensure that write operations do not create an aliasing situation in which a regular file’s contents can be modified through a write to a different regular file.

Figure 5: l2-read-after-write-1

$$\begin{tikzcd}
l2 \arrow[r, "\text{write}"'] \arrow[d, "\text{l2-to-l1-fs}"'] & l2' \arrow[r, "\text{read}"'] \arrow[d, "\text{l2-to-l1-fs}"'] & text \\
l1 \arrow[r, "\text{write}"'] & l1' \arrow[ur, "\text{read}"'] &
\end{tikzcd}$$

These properties, in the form of the predicates `indices-marked-listp` and `no-duplicatesp`, are packaged together into the `l4-stricter-fs-p` predicate, for which a listing follows. Here, the allocation vector `alv` satisfies the type hypothesis `(boolean-listp alv)`.

```
(defun l4-stricter-fs-p (fs alv)
  (declare (xargs :guard t))
  (and (l4-fs-p fs)
       (boolean-listp alv)
       (let ((all-indices (l4-list-all-indices fs)))
            (and (no-duplicatesp all-indices)
                 (indices-marked-p all-indices alv)))))
```

Similarly, for proof purposes we find it useful to package together certain invariants for the stobj `fat32-in-memory`, which we maintain while manipulating the stobj through input/output operations and file operations, in the predicate `compliant-fat32-in-memoryp` for which a listing follows. The constant `*ms-first-data-cluster*`, for instance, is 2, and the last clause of the conjunction helps us maintain an invariant about all cluster indices including the root cluster being greater than or equal to 2, which is stipulated in the FAT32 specification (section 4) and therefore necessary while reasoning about operations in the file allocation table.

```
(defund compliant-fat32-in-memoryp (fat32-in-memory)
  (declare (xargs :stobjs fat32-in-memory :guard t))
  (and (fat32-in-memoryp fat32-in-memory)
       (>= (bpb_bytspersec fat32-in-memory) *ms-min-bytes-per-sector*)
       (>= (bpb_secperclus fat32-in-memory) 1)
       (>= (count-of-clusters fat32-in-memory)
           *ms-fat32-min-count-of-clusters*)
       (>= (bpb_rootclus fat32-in-memory) *ms-first-data-cluster*)))
```

`M2`, however, differs from previous models in that the predicate `compliant-fat32-in-memoryp` cannot be assumed to hold before the filesystem is initialised by reading a disk image. This also means that it is not straightforward to use an abstract stobj [10] for modelling the filesystem state, since the putative invariant `(compliant-fat32-in-memoryp fat32-in-memory)` is not always satisfied.

6.2 Reuse

As noted earlier, in our abstract models, using a refinement methodology allows us to derive our read-over-write properties with little additional effort; more precisely, we are able to prove read-over-write properties simply with `:use` hints after having done the work of proving refinement through induction.

At a lower level, we are also able to benefit from refinement relationships between components of our different models. For example, such a relationship exists between the allocation vector used in `L4` and the file allocation table used in `L6`. More precisely, by taking a file allocation table and mapping each non-zero entry to `true` and each zero entry to `false`, we obtain a corresponding allocation vector with exactly the same amount of available space. This is a refinement mapping which makes it a lot easier to prove that `L4` itself is an abstraction of `L6`. This, in turn, means that the effort spent on proving the invariants described above for `L4` need not be replicated for `L6`.

6.3 The FTY discipline

In the model `M1` in particular, we use the FTY discpline and its associated library [32] to simplify our definitions for regular files, directory files, and other data types. This allows us to simplify as well as speed up our reasoning by eliminating many type hypotheses, and in particular allows us to prove read-over-write properties for `M1` with a significantly smaller number of helper lemmas compared to our abstract models in which FTY is not used.

6.4 Allocation and garbage collection

In the development of our abstract models, `L4` is the first to be bounded in terms of disk size; accordingly, data structures are needed for allocating free blocks on disk, including those which were previously occupied by other files. For simplicity, we choose to model the allocation vector used by the CP/M filesystem [26], which is a boolean array equal in size to the disk recording whether disk blocks are free or in use. Using this, we are able to prove that write operations succeed if and only if there is a sufficient number of free blocks available on the disk, and when we later model FAT32's file allocation table in `L6`, we are able to extend this result by proving a refinement relationship between these two allocation data structures as described earlier.

While it is not straightforward to reuse this result for the file allocation table in `M2`, we are still able to benefit from reusing the `L6` algorithms for disk block allocation and reconstruction of file contents for a given file. As a result, our `L6` proofs for the correctness of these algorithms are also available for reuse in `M2`.

7 Stobjs and co-simulation

Previous work on executable specifications [11] has shown the importance of testing these on real examples, in order to validate that the behaviour shown matches that of the system being specified. In our case, this means we must validate our filesystem by testing it in execution against a canonical implementation of FAT32; in this case, we choose the implementation which ships with Linux kernel 3.10.

For each of our tests, we need to produce FAT32 disk images, which we do with the aid of the program `mkfs.fat` [13]. Further, we make use of this program's verbose mode (enabled through the `-v` command-line switch) for a simple yet fundamental co-simulation test. In the verbose mode, `mkfs.fat` emits an English-language summary of the fields of the newly created disk image; we use `diff` [9] to compare this summary against the output of an ACL2 program, based on our model, which reads the image and pretty-prints the FAT32 fields in the same format. This validates our code for constructing an in-memory representation of the disk image and also serves as a regression test during the process of modifying the model to support proofs and filesystem calls.

Further, we co-simulate `cp` [12], a simple program for copying files, by reproducing its functionality in an ACL2 program. This allows us to validate our code for reading and writing regular files and directories which span multiple clusters.

The quick execution of these co-simulation tests relies upon the use of stobjs. Since the file allocation table and data region (section 4) are large arrays which must be initialised whenever a new disk image is read, ACL2's efficient array operations turn out to be very helpful; this is also the case when we write back a disk image after performing a sequence of file operations. We have also developed a number of special-purpose macros for facilitating the generation of lemmas needed for reasoning about reads and writes to the fields of a stobj in the filesystem context; we expect these to continue to be useful as we work with abstract and concrete stobjs for modelling new filesystems.

8 Conclusion

This work formalises a FAT32-like filesystem and proves read-over-write properties through refinement of a series of models. Further, it proves the correctness of FAT32's allocation and garbage collection mechanisms, and provides artefacts to be used in a subsequent filesystem models.

9 Future work

The FAT32 model is still a work in progress; the set of system calls is not yet complete and the translation functions between disk images, `M2` instances and `M1` instances are not yet verified. However, many of the techniques for these proofs have already been demonstrated in our abstract models. Once we have these, we intend to use them as a basis for reasoning about sequences of filesystem operations in a program, in a manner akin to proving properties of code on microprocessor models. This is a motivation for the pursuit of binary compatibility in our work.

While FAT32 is interesting of and by itself, it lacks features such as crash consistency, which most modern filesystems provide by means of journalling. We hope to reuse some artefacts of formalising FAT32 in order to verify a filesystem with journalling, such as ext4 [24].

We also hope to model the behaviour of filesystems in a multiprogramming environment, where concurrent filesystem calls must be able to occur without corruption or loss of data. Although we have not yet used abstract stobjs for the reasons described above in subsection 6.1, we plan to explore their use in facilitating our stobj reasoning.

Acknowledgments

This material is based upon work supported by the National Science Foundation SaTC program under contract number CNS-1525472. Thanks are also owed to Warren A. Hunt Jr. and Matt Kaufmann for their guidance, and to the anonymous reviewers for their suggestions.

References

[1] Martín Abadi & Leslie Lamport (1991): *The existence of refinement mappings*. Theoretical Computer Science 82(2), pp. 253–284, doi:10.1016/0304-3975(91)90224-P.

[2] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell et al. (2016): *Cogent: Verifying high-assurance file system implementations*. In: *ACM SIGPLAN Notices*, 51, ACM, pp. 175–188, doi:10.1145/2872362.2872404.

[3] Yves Bertot & Pierre Castéran (2013): *Interactive theorem proving and program development: CoqArt: the calculus of inductive constructions*. Springer Science & Business Media, doi:10.1007/978-3-662-07964-5.

[4] William R. Bevier & Richard M. Cohen (1996): *An executable model of the Synergy file system*. Technical Report, Technical Report 121, Computational Logic, Inc.

[5] Robert S. Boyer & J Strother Moore (2002): *Single-threaded objects in ACL2*. In: *International Symposium on Practical Aspects of Declarative Languages*, Springer, pp. 9–27, doi:10.1007/3-540-45587-6_3.

[6] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek & Nickolai Zeldovich (2016): *Using Crash Hoare Logic for Certifying the FSCQ File System*. In Ajay Gulati & Hakim Weatherspoon, editors: *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016.*, USENIX Association, doi:`10.1145/2815400.2815402`. Available at `https://www.usenix.org/conference/atc16`.

[7] Russ Cox, M. Frans Kaashoek & Robert T. Morris: *Xv6, a simple Unix-like teaching operating system, 2016*. Available at `http://pdos.csail.mit.edu/6.828/2014/xv6.html`.

[8] Leonardo De Moura & Nikolaj Bjørner (2008): *Z3: An efficient SMT solver*. In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, pp. 337–340, doi:`10.1007/978-3-540-78800-3_24`.

[9] Paul Eggert, Mike Haertel, David Hayes, Richard Stallman & Len Tower: *diff (1)-Linux manual page, accessed: 07 Sep 2018*.

[10] Shilpi Goel, Warren A. Hunt Jr. & Matt Kaufmann (2013): *Abstract stobjs and their application to ISA modeling*. *arXiv preprint arXiv:1304.7858*, doi:`10.4204/EPTCS.114.5`.

[11] 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*, IEEE, pp. 91–98, doi:`10.1109/FMCAD.2014.6987600`.

[12] Torbjorn Granlund, David MacKenzie & Jim Meyering: *cp (1)-Linux manual page, accessed: 09 Jul 2018*.

[13] Dave Hudson, Peter Anvin & Roman Hodek: *mkfs.fat (8)-Linux manual page, accessed: 09 Jul 2018*.

[14] Matt Kaufmann, Panagiotis Manolios & J Strother Moore (2000): *Computer-aided reasoning: an approach*. Kluwer Academic Publishers, doi:`10.1007/978-1-4757-3188-0`.

[15] Michael Kerrisk: *close (2)-Linux manual page, accessed: 09 Jul 2018*.

[16] Michael Kerrisk: *errno (3)-Linux manual page, accessed: 07 Sep 2018*.

[17] Michael Kerrisk: *lstat (2)-Linux manual page, accessed: 09 Jul 2018*.

[18] Michael Kerrisk: *mkdir (2)-Linux manual page, accessed: 09 Jul 2018*.

[19] Michael Kerrisk: *mknod (2)-Linux manual page, accessed: 09 Jul 2018*.

[20] Michael Kerrisk: *open (2)-Linux manual page, accessed: 09 Jul 2018*.

[21] Michael Kerrisk: *pread (2)-Linux manual page, accessed: 09 Jul 2018*.

[22] Michael Kerrisk: *pwrite (2)-Linux manual page, accessed: 09 Jul 2018*.

[23] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish et al. (2009): *seL4: Formal verification of an OS kernel*. In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ACM, pp. 207–220, doi:`10.1145/1629575.1629596`.

[24] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas & Laurent Vivier (2007): *The new ext4 filesystem: current status and future plans*. In: *Proceedings of the Linux symposium*, 2, pp. 21–33.

[25] Microsoft (2000): *Microsoft Extensible Firmware Initiative FAT32 File System Specification*. Available at `https://download.microsoft.com/download/1/6/1/161ba512-40e2-4cc9-843a-923143f3456c/fatgen103.doc`.

[26] Michael Moria: *cpm (5)-Linux manual page, accessed: 07 Sep 2018*.

[27] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak & Xi Wang (2017): *Hyperkernel: Push-Button Verification of an OS Kernel*. In: *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, ACM, New York, NY, USA, pp. 252–269. Available at `http://doi.acm.org/10.1145/3132747.3132748`.

[28] Tobias Nipkow, Lawrence C. Paulson & Markus Wenzel (2002): *Isabelle/HOL: a proof assistant for higher-order logic*. 2283, Springer Science & Business Media, doi:`10.1007/3-540-45949-9`.

[29] Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy & Peter Sewell (2015): *SibylFS: formal specification and oracle-based testing for POSIX and real-world file systems*. In: *Proceedings of the 25th Symposium on Operating Systems Principles*, ACM, pp. 38–53, doi:`10.1145/2815400.2815411`.

[30] Peter Jay Salzman & Ori Pomerantz (2001): *The Linux Kernel Module Programming Guide*, chapter 4. Available at `https://www.tldp.org/LDP/lkmpg/2.4/html/c577.htm`.

[31] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak & Xi Wang (2016): *Push-Button Verification of File Systems via Crash Refinement*. In: *OSDI*, 16, pp. 1–16. Available at `https://www.usenix.org/conference/atc17/technical-sessions/presentation/sigurbjarnarson`.

[32] Sol Swords & Jared Davis (2015): *Fix Your Types*. In: *Proceedings of the Thirteenth International Workshop on the ACL2 Theorem Prover and its Applications (ACL2 '15)*, doi:`10.4204/EPTCS.192.2`.

Trapezoidal Generalization over Linear Constraints

David Greve
david.greve@rockwellcollins.com

Andrew Gacek
andrew.gacek@rockwellcollins.com

We are developing a model-based fuzzing framework that employs mathematical models of system behavior to guide the fuzzing process. Whereas traditional fuzzing frameworks generate tests randomly, a model-based framework can deduce tests from a behavioral model using a constraint solver. Because the state space being explored by the fuzzer is often large, the rapid generation of test vectors is crucial. The need to generate tests quickly, however, is antithetical to the use of a constraint solver. Our solution to this problem is to use the constraint solver to generate an initial solution, to generalize that solution relative to the system model, and then to perform rapid, repeated, randomized sampling of the generalized solution space to generate fuzzing tests. Crucial to the success of this endeavor is a generalization procedure with reasonable size and performance costs that produces generalized solution spaces that can be sampled efficiently. This paper describes a generalization technique for logical formulae expressed in terms of Boolean combinations of linear constraints that meets the unique performance requirements of model-based fuzzing. The technique represents generalizations using *trapezoidal* solution sets consisting of ordered, hierarchical conjunctions of linear constraints that are more expressive than simple intervals but are more efficient to manipulate and sample than generic polytopes. Supporting materials contain an ACL2 proof that verifies the correctness of a low-level implementation of the generalization algorithm against a specification of generalization correctness. Finally a post-processing procedure is described that results in a restricted trapezoidal solution that can be sampled (solved) rapidly and efficiently without backtracking, even for integer domains. While informal correctness arguments are provided, a formal proof of the correctness of the restriction algorithm remains as future work.

1 Motivation

Fuzzing is a form of robustness testing in which random, invalid or unusual inputs are applied while monitoring the overall health of the system. Model-based fuzzing is a fuzzing technique that employs a mathematical model of system behavior to guide the fuzzing process and explore behaviors that would be difficult to reach by chance. Whereas many fuzzing frameworks generate tests randomly, a model-based framework can deduce tests from a behavioral model using a constraint solver. Because the state space being explored by the fuzzer is generally large, the rapid generation of test vectors is crucial. Unfortunately, the need to generate tests quickly is antithetical to the use of a constraint solver. Our solution to this problem is to use a constraint solver to generate an initial solution and then to generalize that solution relative to the constraint. Test generation in our model-based fuzzing framework, therefore, consists of repeated, randomized sampling of the generalized solution space. Generalization is crucial to the performance of our framework because it allows us to decouple constraint solving (slow) from test generation (fast).

The fuzzing algorithm, outlined in Algorithm 1, begins by identifying an appropriate logical constraint, selected according to some testing heuristic. A constraint solver then attempts to generate a

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA) under DARPA/AFRL Contract FA8750-16-C-0218. The views, opinions and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Approved for Public Release, Distribution Unlimited

Algorithm 1: Conceptual Model-Based Fuzzing Algorithm

```
while $True$ do
    $L \longleftarrow nextConstraint()$
    $\vec{v}, SAT \longleftarrow solve(L)$
    if $SAT$ then
        $T \longleftarrow generalize(L, \vec{v})$
        $T', \sigma \longleftarrow restrict(T, \vec{v})$
        for $A\ While$ do
            $\vec{w} \longleftarrow sample(T', \sigma)$
            $fuzzTarget(\vec{w})$
        end
    end
end
```

variable assignment that satisfies the constraint. If it succeeds, the solution is generalized relative to the constraint. The generalization is then restricted so that it can be sampled efficiently. The restricted generalization is then repeatedly sampled to generate random vectors (known to satisfy the constraint) that are then applied to the fuzzing target. This process is repeated for the duration of the fuzzing session.

Trapezoidal generalization is uniquely suited for use in model-based fuzzing. The trapezoidal generalization process is reasonably efficient, both in terms of speed and the size of the final representation. Unlike interval generalization, trapezoidal generalization is capable of representing many linear model features exactly, allowing sampled tests to better target relevant model behaviors. Finally, unlike generic polytope generalization, trapezoidal generalization supports efficient sampling of the solution space, enabling rapid test generation and addressing the performance requirements of model-based fuzzing.

This paper provides details on our trapezoidal generalization, restriction, and sampling algorithms and the supporting material includes an ACL2 proof of the correctness of a low-level implementation of generalization. Section 2 describes the trapezoidal data structure and show how it can be used to generalize solutions relative to logical formulae expressed as Boolean combinations of linear constraints. It includes a formal specification of generalization correctness and an outline of crucial aspects of the correctness proof for our generalization algorithm. Section 3 describes sampling and provides details of the restriction process that refines a trapezoid so that it can be rapidly and efficiently sampled (solved) without backtracking, even for integer domains. While informal correctness arguments are provided, a formal proof of the correctness of the restriction algorithm remains as future work. Section 4 provides background on our implementation and formalization of trapezoidal generalization and Section 5 compares our experience using both trapezoidal and interval generalization and provides an overview of related work in the field.

1.1 Problem Syntax

Let $x_1, \ldots, x_n$ be variables of mixed integer and rational types. Each variable has an associated numeric *dimension*, denoted by the variable subscript, that establishes a complete ordering among the variables. Numeric expressions are represented as polynomials (P) over variables having the form $c_n x_n + \cdots c_1 x_1 + c_0$ where each c_i is a rational constant. We specifically restrict our attention to linear, rational, multivariate polynomials.

$$P \equiv c_n x_n + \cdots c_1 x_1 + c_0$$

The largest k for which c_k is non-zero is called the dimension of a polynomial. We sometimes write P_k to explicitly identify the dimension of a polynomial, $\dim(P_k) = k$. Note that constants have dimension 0.

A vector is a sequence of values ordered by dimension. Given a vector $\vec{v} = v_n, \ldots, v_1$, we write $E[\vec{v}]$ to denote the evaluation of an expression E where each x_i in that expression is replaced by v_i. We assume that every variable has an associated value in the vector and consider only *consistent vectors*, vectors where each dimension's value is consistent with it's corresponding variable's type, either rational or integral. Constants are self-evaluating and the evaluation operation distributes over the standard operations in the expected ways.

The atoms in our formulae are linear constraints. A linear constraint (L) is an equality or inequality over polynomials. We will often write $\prec$ to stand for either $<$ or $\leq$ and $\succ$ to stand for either $>$ or $\geq$.

$$L \equiv P_i = P_j \mid P_i > P_j \mid P_i \geq P_j \mid P_i < P_j \mid P_i \leq P_j$$

Logical formulae (F) are defined over linear constraints using conjunction, disjunction, and negation.

$$F \equiv F \wedge F \mid F \vee F \mid \neg F \mid L$$

We define a variable *bound* (B) as a linear constraint on a single variable. The dimension of a variable bound is the dimension of the bound variable. We say that a variable bound is *normalized* if the dimension of the variable is greater than the dimension of the bounding polynomial, $n > m$.

$$B_n \equiv (x_n \prec P_m) \mid (x_n \succ P_m) \mid (x_n = P_m)$$

We define a *trapezoid* (T) as a (possibly empty) set of variable bounds. A trapezoid can be interpreted logically as the conjunction of all of its variable bounds or geometrically as the intersection of the volumes defined by the planes bounding each variable. We say that a vector *satisfies* a trapezoid if it is a consistent vector and the logical interpretation of the trapezoid evaluates to True at the vector. A trapezoid is *satisfiable* if a consistent vector exists for which it evaluates to True. Equivalently, we can say that a trapezoid *contains* a vector if the point defined by that vector resides inside of the volumes defined by its geometric interpretation. A satisfiable trapezoid contains at least one point. Below is a grammar for trapezoids expressed in terms of the intersection of variable bounds.

$$T \equiv \; B \cap T \mid B \mid \varnothing$$

In addition the above syntactic restrictions, we also require that the constituent bounds of a trapezoid satisfy the following three properties:

1. All variable bounds must be normalized
2. The set of bounds must be simultaneously satisfiable as witnessed by a consistent reference vector $\vec{v}$
3. Each x_n is either unbound or is bound by either a single equality or at most one upper bound and at most one lower bound.

We call this data structure a trapezoid because, for every dimension greater than one, the bounds on the variable of that dimension are, in general, linear relations expressed in terms of the preceding variable. Holding all other variables constant, such linear bounds describe trapezoidal regions; hence the name.

The data structure we use to represent generalizations is a *region*. A region is defined as either a trapezoid or the complement of a trapezoid:

$$R \equiv \{T\} \mid {}^{\sim}\{T\}$$

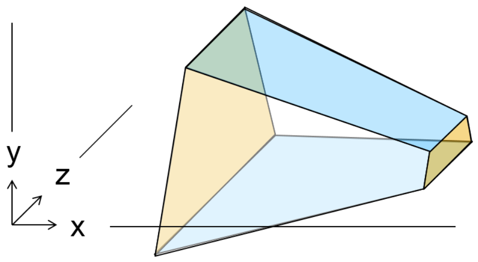


Figure 1: 3-d Trapezoidal Volume

2 Generalization

An ACL2 proof that a low-level implementation of the following generalization procedure satisfies our notion of generalization correctness is provided in the supporting materials accompanying this paper. The following presentation provides an overview of the generalization process but limits discussion of correctness to observations concerning key steps in this proof along with references to the supporting materials.

2.1 Generalization Correctness

Given a formula F and a consistent reference vector $\vec{v} = v_n, \ldots, v_1$, the objective is to compute a region R which correctly generalizes $\vec{v}$ with respect to F. We say that a generalization is correct if it satisfies the following properties:

- **Invariant 1** $F[\vec{v}] = R[\vec{v}]$
- **Invariant 2.a** If $F[\vec{v}]$, then $\forall \vec{w}.\ R[\vec{w}] \Rightarrow F[\vec{w}]$
- **Invariant 2.b** If not $F[\vec{v}]$, then $\forall \vec{w}.\ F[\vec{w}] \Rightarrow R[\vec{w}]$

The first invariant ensures that the reference vector is contained in the generalization iff the vector satisfies the original formula. The second two invariants ensure that the generalized result is a *conservative under-approximation* of the original formula. Together they guarantee that the state space that contains the reference vector in the generalized result is a subset of the state space that contains the reference vector in the original formula.

2.2 Generalization Process

The generalization process involves transforming a reference vector and a logical formula into a trapezoidal region. Generalizing relative to linear formula (Section 2.6) begins by transforming linear relations into regions (Section 2.3) and it proceeds by alternately intersecting and complementing the resulting regions (Section 2.5). The process of intersecting regions may further involve generalizing the intersection of sets of linear bounds into trapezoids (Section 2.4). The rules for each of these operations is presented in a bottom-up fashion.

2.3 Generalizing Linear Relations

We use $\langle\langle L\rangle\rangle \xrightarrow{\vec{v}} R$ to represent the generalization of a linear relation (L) into a trapezoidal region (R), a process which may involve several steps. First, arbitrary linear relations between two polynomials can be expressed as linear relations between a polynomial and zero.

$$\langle\langle P_i \prec P_j\rangle\rangle \xrightarrow{\vec{v}} \langle\langle P_i - P_j \prec 0\rangle\rangle \quad \langle\langle P_i \succ P_j\rangle\rangle \xrightarrow{\vec{v}} \langle\langle P_j - P_i \prec 0\rangle\rangle \quad \langle\langle P_i = P_j\rangle\rangle \xrightarrow{\vec{v}} \langle\langle P_i - P_j = 0\rangle\rangle$$

Constant linear relations normalize into either the true (empty) trapezoid or its complement.

$$\begin{array}{ll}
\langle\langle c_0 \prec 0\rangle\rangle \xrightarrow{\vec{v}} \{\} \quad \text{if } c_0 \prec 0 & \langle\langle c_0 = 0\rangle\rangle \xrightarrow{\vec{v}} \{\} \quad \text{if } c_0 = 0 \\
\langle\langle c_0 \prec 0\rangle\rangle \xrightarrow{\vec{v}} {\sim}\{\} \quad \text{if } \neg(c_0 \prec 0) & \langle\langle c_0 = 0\rangle\rangle \xrightarrow{\vec{v}} {\sim}\{\} \quad \text{if } \neg(c_0 = 0)
\end{array}$$

Non-constant linear relations between polynomials and zero are normalized so that they relate the variable with the largest dimension to a polynomial consisting only of smaller variables and constants.

$$\begin{array}{ll}
\langle\langle P_n < 0\rangle\rangle \xrightarrow{\vec{v}} \langle\langle x_n < -(P_n/c_n - x_n)\rangle\rangle & \text{if } c_n > 0 \\
\langle\langle P_n < 0\rangle\rangle \xrightarrow{\vec{v}} \langle\langle -(P_n/c_n - x_n) < x_n\rangle\rangle & \text{if } c_n < 0 \\
\langle\langle P_n \le 0\rangle\rangle \xrightarrow{\vec{v}} \langle\langle x_n \le -(P_n/c_n - x_n)\rangle\rangle & \text{if } c_n > 0 \\
\langle\langle P_n \le 0\rangle\rangle \xrightarrow{\vec{v}} \langle\langle -(P_n/c_n - x_n) \le x_n\rangle\rangle & \text{if } c_n < 0 \\
\langle\langle P_n = 0\rangle\rangle \xrightarrow{\vec{v}} \langle\langle x_n = -(P_n/c_n - x_n)\rangle\rangle &
\end{array}$$

Normalized linear relations that are true at the reference vector simply become singleton trapezoidal regions.

$$\begin{array}{ll}
\langle\langle x_n \prec P\rangle\rangle \xrightarrow{\vec{v}} \{(x_n \prec P)\} & \text{if } x_n[\vec{v}] \prec P[\vec{v}] \\
\langle\langle P \prec x_n\rangle\rangle \xrightarrow{\vec{v}} \{(P \prec x_n)\} & \text{if } P[\vec{v}] \prec x_n[\vec{v}] \\
\langle\langle x_n = P\rangle\rangle \xrightarrow{\vec{v}} \{(x_n = P)\} & \text{if } x_n[\vec{v}] = P[\vec{v}]
\end{array}$$

A normalized linear relation that evaluates to false at the reference vector, however, is expressed as a negated trapezoidal region containing a single linear relation that is true at the reference vector.

$$\begin{array}{lll}
\langle\langle x_n < P\rangle\rangle \xrightarrow{\vec{v}} {\sim}\{(P \le x_n)\} & \text{if } \neg(x_n[\vec{v}] < P[\vec{v}]) & \\
\langle\langle x_n \le P\rangle\rangle \xrightarrow{\vec{v}} {\sim}\{(P < x_n)\} & \text{if } \neg(x_n[\vec{v}] \le P[\vec{v}]) & \\
\langle\langle P < x_n\rangle\rangle \xrightarrow{\vec{v}} {\sim}\{(x_n \le P)\} & \text{if } \neg(x_n[\vec{v}] > P[\vec{v}]) & \\
\langle\langle P \le x_n\rangle\rangle \xrightarrow{\vec{v}} {\sim}\{(x_n < P)\} & \text{if } \neg(x_n[\vec{v}] \ge P[\vec{v}]) & \\
\langle\langle x_n = P\rangle\rangle \xrightarrow{\vec{v}} {\sim}\{(P < x_n)\} & \text{if } (x_n[\vec{v}] > P[\vec{v}]) & \text{(neq.1)} \\
\langle\langle x_n = P\rangle\rangle \xrightarrow{\vec{v}} {\sim}\{(x_n < P)\} & \text{if } (x_n[\vec{v}] < P[\vec{v}]) & \text{(neq.2)}
\end{array}$$

Normalizing linear relations in this way ensures that every variable bound contained in the body of a trapezoid is true at the reference vector. It also ensures that regions that contain the reference vector simplify into simple trapezoids while those that do not are expressed as the complement of a trapezoid that does.

With the exception of rules neq.1 and neq.2, the rules for generalizing linear relations ensure that the generalization is equal to the original formula (i.e. $\forall w.\ F[\vec{w}] = L[\vec{w}]$), trivially satisfying our correctness invariants. Crucially, rules neq.1 and neq.2 do satisfy Invariants 1 and 2.b (and, trivially, 2.a) as expressed in Section 2.1, ensuring that, in all cases, our generalization of linear constraints is correct. See the functions `normalize-equal-0` and `normalize-gt-0` and their associated lemmas in the file `top.lisp` of the supporting materials.

2.4 Generalizing Bound Intersections

Let C be a set of normalized bounds known to be satisfiable as witnessed by the consistent reference vector $\vec{v}$. The relation $C \xrightarrow{\vec{v}}{}^{*} T$ represents the fixed-point of intersecting and reducing the various bounds contained in C into a trapezoidal region. The following collection of rules govern the individual intersections of the members of such a set of bounds to produce a trapezoid. The rules of the rewrite system operate on an un-ordered set of bounds, allowing for arbitrary reordering of the bounds. The key to this process are rules for eliminating multiple bounds on the same variable. The rules for eliminating multiple upper bounds are shown below. The rules for multiple lower bounds are symmetric.

$$(x_n < P) \cap (x_n < Q) \xrightarrow{\vec{v}} (x_n < P) \cap \langle\langle P \leq Q \rangle\rangle \qquad \text{if } P[\vec{v}] \leq Q[\vec{v}] \qquad \text{(lt-int.1)}$$

$$(x_n < P) \cap (x_n \leq Q) \xrightarrow{\vec{v}} (x_n < P) \cap \langle\langle P \leq Q \rangle\rangle \qquad \text{if } P[\vec{v}] \leq Q[\vec{v}] \qquad \text{(lt-int.2)}$$

$$(x_n \leq P) \cap (x_n < Q) \xrightarrow{\vec{v}} (x_n \leq P) \cap \langle\langle P < Q \rangle\rangle \qquad \text{if } P[\vec{v}] < Q[\vec{v}] \qquad \text{(lt-int.3)}$$

$$(x_n \leq P) \cap (x_n \leq Q) \xrightarrow{\vec{v}} (x_n \leq P) \cap \langle\langle P \leq Q \rangle\rangle \qquad \text{if } P[\vec{v}] \leq Q[\vec{v}] \qquad \text{(lt-int.4)}$$

Note that while $\langle\langle P \prec Q \rangle\rangle$ normalizes the linear relation as described in Section 2.3 this bound will never evaluate to false due to the rules' side-conditions. Consequently the result will always be a simple trapezoid which is either empty (if both bounds are constant) or contains a single normalized bound on a variable whose dimension is less than that of x_n. Similar observations apply to the rules for simplifying bounds involving equality, as shown below. This process and it's correctness is captured by the function `andTrue-variableBound-variableBound` and its associated lemmas (generated by the `def::trueAnd` macro) in the file `poly-proofs.lisp` of the supporting materials.

$$(x_n = P) \cap (x_n \prec Q) \xrightarrow{\vec{v}} (x_n = P) \cap \langle\langle P \prec Q \rangle\rangle \qquad \text{(eq-int.1)}$$

$$(x_n = P) \cap (Q \prec x_n) \xrightarrow{\vec{v}} (x_n = P) \cap \langle\langle Q \prec P \rangle\rangle \qquad \text{(eq-int.2)}$$

$$(x_n = P) \cap (x_n = Q) \xrightarrow{\vec{v}} (x_n = P) \cap \langle\langle P = Q \rangle\rangle \qquad \text{(eq-int.3)}$$

Lemma 2.1. *The intersection rules are terminating.*

Proof. Define a measure on the conjunction of bounds to be the sum of the dimension of each bound. Each rule of the intersection rewrite system reduces this measure. For example, consider rule lt-int.1. We know $dim(x) > dim(P)$ and $dim(x) > dim(Q)$, and thus $dim(x) > dim(\langle P \leq Q \rangle)$ and in particular $dim(x < Q) > dim(\langle P \leq Q \rangle)$. Since the measure is always non-negative, the intersection rules are terminating. See the `def::total` event for the function `intersect` in the file `intersection.lisp` of the supporting materials for a proof of termination for ordered (not un-ordered, see Section 3.0.1) bounds. □

Lemma 2.2. *When run to completion, the intersection rules result in a trapezoid.*

Proof. The intersection rules apply to a set of bounds that are normalized and satisfiable. To qualify as a trapezoid the only missing requirement is that there may be multiple upper, lower, and equality bounds on each variables. For multiple upper bounds, note that the 4 rules above cover all cases. Although rule lt-int.3 does not cover the case when $P[\vec{v}] = Q[\vec{v}]$, that case will in fact be covered by rule lt-int.2 since then $Q[\vec{v}] \leq P[\vec{v}]$. The case of multiple lower bounds is symmetric. Therefore, the rewrite rules will eliminate all multiple and lower and upper bounds so that the result is a trapezoid. See lemma `trapezoid-p-intersect` in the file `intersection.lisp` of the supporting materials. □

While, in general, the the fixed-point resulting from intersecting and reducing a set of bounds will differ from the original set, each rule that we apply to perform this normalization satisfies Invariants 1 and 2.a (and trivially 2.b) as expressed in Section 2.1, ensuring the correctness of this process.

2.5 Generalizing Region Intersections and Complements

The relation $R^a \cap R^b \overset{\vec{v}}{\rightarrowtail} R^c$ indicates that region R^c is a generalized intersection of regions R^a and R^b relative to $\vec{v}$ and $\sim R^a \overset{\vec{v}}{\rightarrowtail} R^b$ to indicate that R^b is the generalized complement of R^a. We characterize the behavior of region intersection and complement with the following set of rules:

$$\frac{}{\sim\sim\{T\} \overset{\vec{v}}{\rightarrowtail} \{T\}}\ \textsc{comp} \qquad \frac{T^a \cap T^b \xrightarrow{\vec{v}}{}^{*}\ T^c}{\{T^a\} \cap \{T^b\} \overset{\vec{v}}{\rightarrowtail} \{T^c\}}\ \textsc{tint}$$

$$\frac{}{\sim\{T\} \cap R \overset{\vec{v}}{\rightarrowtail} \sim\{T\}}\ \textsc{cint}.1 \qquad \frac{}{R \cap \sim\{T\} \overset{\vec{v}}{\rightarrowtail} \sim\{T\}}\ \textsc{cint}.2$$

While perhaps surprising, the rules CINT.1 and CINT.2 not only simplify the region intersection process, they are also both consistent with and essential for preserving generalization correctness (specifically, Invariant 2.b) as described in Section 2.1. See the functions `and-regions` and `not-region` and their associated lemmas in the file `top.lisp` from the supporting materials.

2.6 Generalizing Linear Formulae

We write $F \overset{\vec{v}}{\Rightarrow} R$ to mean that region R generalizes $\vec{v}$ with respect to formula F. In general, conjunction generalizes into region intersection, negation generalizes into region complement, and disjunction generalizes into a complement of the intersection of the complements of the generalized arguments. The generalization relation is defined inductively over the structure of the formula F.

$$\frac{\langle\langle L \rangle\rangle \xrightarrow{\vec{v}} R}{L \overset{\vec{v}}{\Rightarrow} R}\ \textsc{gen-atom} \qquad \frac{F_1 \overset{\vec{v}}{\Rightarrow} R_1 \quad F_2 \overset{\vec{v}}{\Rightarrow} R_2}{F_1 \wedge F_2 \overset{\vec{v}}{\Rightarrow} R_1 \cap R_2}\ \textsc{gen-and}$$

$$\frac{F \overset{\vec{v}}{\Rightarrow} R}{\neg F \overset{\vec{v}}{\Rightarrow} {\sim}R}\ \text{GEN-NOT} \qquad \frac{F_1 \overset{\vec{v}}{\Rightarrow} R_1 \quad F_2 \overset{\vec{v}}{\Rightarrow} R_2}{F_1 \vee F_2 \overset{\vec{v}}{\Rightarrow} {\sim}({\sim}R_1 \cap {\sim}R_2)}\ \text{GEN-OR}$$

This procedure can be shown to satisfy our correctness invariants by induction over the structure of a linear formula and by appealing to the correctness of the generalization of linear relations and region intersection. See the definition `generalize-ineq` and the theorems `inv1-generalize-ineq` and `inv2-generalize-ineq` in the file `top.lisp` of the supporting materials.

3 Sampling

Sampling is the process of identifying randomized, concrete, type-consistent variable assignments (vectors) that satisfy all of the variable bounds in a trapezoid [1]. We call this process sampling, rather than solving, to emphasize the fact that it is intended to be randomized, simple, and efficient. Indeed, the structure of a trapezoid lends itself to a simple and efficient sampling algorithm. The trapezoidal structure ensures that the variable with the smallest dimension is bounded only by constants[2]. Sampling of the smallest dimension variable therefore involves simply choosing a value consistent with its type and constant bounds. Because each normalized variable bound is expressed strictly in terms of variables of smaller dimension, sampling the variables by proceeding from the smallest to the largest dimension guarantees that the bounds on each subsequent variable will evaluate to constants relative to variables already assigned. Sampling each variable from the smallest to the largest dimension, therefore, involves nothing more than choosing a value for each variable consistent with type and constant bounds computed based on the previous variable assignments.

While the above algorithm is certainly simple and efficient, the trapezoidal data structure as defined is not sufficient to guarantee that every variable assignment that satisfies the first N bounds will satisfy the $N+1^{th}$ bound. A given set of variable assignments may result in an upper bound that is smaller than a lower bound or, more subtly, an integer variable may be constrained by bounds that do not permit an integral solution. While backtracking and attempting different sets of variable assignments is possible in such cases, it can also be very inefficient.

To address this concern we present an overview of a post-processing step for trapezoids, performed before sampling, that results in a restricted trapezoid with properties that ensure that the simple and efficient sampling algorithm presented above will never encounter a bound that cannot be satisfied (even for integer variables) and thus never needs to backtrack. While the following procedure has not been mechanically verified, we sketch selected aspects of the informal proof with the long term objective of formalizing and verifying its correctness in ACL2.

3.0.1 Ordered Trapezoids

To better explain our post-processing approach we first introduce a more refined representation for trapezoids, beginning with the concept of variable *intervals*. A variable interval is either a single variable bound or a pair of bounds consisting of an upper and lower bound on the same variable. The dimension of an interval is simply the dimension of the bound variable. Variable intervals allow us to gather all of the linear bounds relevant to each variable in one place.

[1]We focus on trapezoids because solving a trapezoid complement is trivial. The complement of a trapezoid is merely a disjunction of negated linear bounds. An assignment that satisfies any one of the negated bounds satisfies the trapezoid complement.

[2]Technically, variables may also be unbound. In such case arbitrary value assignments suffice.

$$I_n \equiv B_n \mid (P^L \prec x_n) \cap (x_n \prec P^U)$$

Variable intervals can be used to construct ordered trapezoids. An ordered trapezoid is simply a sequence of variable intervals organized in descending order based on dimension, terminating in the empty trapezoid. Note that these new syntactic constructs do not change in any way the nature of trapezoids other than to impose additional ordering constraints on their representation.

$$\begin{aligned} T_n &\equiv \; I_n \cap T_m \;\; \text{where } n > m \\ T_0 &\equiv \varnothing \end{aligned}$$

Using ordered trapezoids, we can define the process for sampling a trapezoid, $\varepsilon(T)$ to produce a consistent vector that satisfies the trapezoid. Note that $\varepsilon(I_n[\vec{w}])$ represents a random choice of a value consistent in type with variable n and satisfying the interval I_n evaluated at the vector $\vec{w}$ (assuming such a value exists), $\{\}$ represents an empty vector, and $\vec{w}[i] = x$ represents an update of vector $\vec{w}$ such that dimension i is equal to the value x.

$$\frac{}{\varepsilon(\varnothing) \rightarrow \{\}} \qquad\qquad \frac{\varepsilon(T_m) \rightarrow \vec{w}}{\varepsilon(I_n \cap T_m) \rightarrow \vec{w}[n] = \varepsilon(I_n[\vec{w}])}$$

3.1 Restriction

The purpose of post-processing is to restrict the domain of values that can be assigned to selected variables in the trapezoid to ensure that the sampling process can proceed without backtracking. We call the trapezoidal post-processing step *restriction*. The restriction process takes as input a trapezoid, a change of basis (σ, initially an identity function), and a consistent reference vector. The output of the restriction process is a new (restricted) trapezoid and a change of basis that maps vectors sampled from the restricted trapezoid back into the vector space of the original trapezoid.

$$T, \sigma, \vec{v} \xrightarrow{R} T', \sigma'$$

The restriction process is specified as a single pass over the trapezoid intervals, starting with the largest interval and proceeding to the smallest. Each interval in the trapezoid is restricted by applying the steps summarized below.

- **Bound Fixing (BF)** Ensures that linear bounds are expressed only in terms of $\geq$ and $\leq$.
- **Integer Equality (IE)** Ensures that equalities involving integer variables are always satisfiable.
- **Interval Restriction (IR)** Ensures that intervals are always satisfiable.

Each interval in the trapezoid may suggest a domain restriction, in the form of a set of linear constraints, and a change of basis in the form of a linear transformation. Domain restrictions are intersected with the remaining trapezoid before it, too, is restricted. Changes of base are accumulated and applied to the current interval, the remaining trapezoid, and the reference vector. When complete, the final restricted trapezoid and the final change of basis are returned.

$$\frac{}{\varnothing,\, \sigma,\, \vec{v} \xrightarrow{R} \varnothing,\, \sigma}\ \textsc{Restriction-Base}$$

$$\frac{\begin{array}{c} I_n \cap T_m,\, \sigma,\, \vec{v} \xrightarrow{BG} I'_n \cap T'_m,\, \sigma',\, \vec{v}' \\ I'_n \cap T'_m,\, \sigma',\, \vec{v}' \xrightarrow{IE} I''_n \cap T''_m,\, \sigma'',\, \vec{v}'' \\ I''_n \cap T''_m,\, \sigma'',\, \vec{v}'' \xrightarrow{IR} I'''_n \cap T'''_m,\, \sigma''',\, \vec{v}''' \\ T'''_m,\, \sigma''',\, \vec{v}''' \xrightarrow{R} T''''_m,\, \sigma'''' \end{array}}{I_n \cap T_m,\, \sigma,\, \vec{v} \xrightarrow{R} I'''_n \cap T''''_m,\, \sigma'''}\ \textsc{Restriction-Step}$$

The restricted trapezoid can then be sampled using our simple sampling algorithm and the resulting vectors can be mapped back into the original vector space using the change of basis in such a way that $\sigma'[\varepsilon(T')]$ is a type consistent vector and $T[\sigma'[\varepsilon(T')]]$ is true.

The following discussion assumes that all integer valued variables are bounded only by other integer valued variables. This can be ensured if, say, the dimension of each rational variable is always greater than the dimension of every integer variable.

3.2 Bound Fixing (BF)

Bound fixing transforms exclusive inequalities on integer variables into inclusive inequalities and tightens any remaining integer bounds.

$$\frac{BF(I_n) \to I'_n}{I_n \cap T_m,\, \sigma,\, \vec{v} \xrightarrow{BF} I'_n \cap T_m,\, \sigma,\, \vec{v}}\ \textsc{Bound-Fixing}$$

We omit a more detailed description of this operation for space considerations.

3.3 Integer Equality (IE)

Any integer variable equality bound appearing in a trapezoid can be expressed as:

$$x_n = (N_k * x_k + \ldots + N_0)/D_n$$

where all x_i and N_i are integer valued and D_n is a positive integer, representing the least common denominator of the bounding polynomial coefficients. Given $a, b \in \mathbb{Z}$ we write $(a|b)$, called "a divides b", if and only if there exists $c \in \mathbb{Z}$ such that $b = a * c$. A divisibility constraint is an expression of the form $(m|E)$ where $m \in \mathbb{Z}$ and $E \in \mathscr{E}$. A vector $\vec{v}$ is called a *solution* to a divisibility constraint $(m|E)$ if and only if $(m|E[\vec{v}])$. A divisibility constraint is called solvable if and only if it has at least one solution. To ensure that x_n has an integer solution, $\vec{w}$ must be chosen so that $(D_n|(N_k * x_k + \ldots + N_0)[\vec{w}])$. Of course if D_n is 1, this is trivial and no additional work is required.

Given a non-trivial, solvable divisibility constraint $(m|E)$ we want to find an efficient way to enumerate all solutions. Towards this end, we introduce the concept of a *Trapezoidal Change of Basis* (TCOB): a variable substitution that preserves dimension.

Definition 3.1 (Trapezoidal Change of Basis)**.** *A map $\sigma : \mathscr{E} \to \mathscr{E}$ is called an trapezoidal change of basis, if*

1. $\sigma(c_0 + c_1x_1 + \cdots + c_nx_n) = c_0 + c_1\sigma(x_1) + \cdots + c_n\sigma(x_n)$, *i.e.* σ *is a homomorphism, and*
2. $\dim(\sigma(x_i)) = \dim(x_i)$, *for all* $i = 1, \ldots, n$.

For a solvable divisibility constraint $(m|E)$, we propose to construct a TCOB σ such that 1) every vector is a solution to the divisibility constraint $(m|\sigma(E))$ and 2) each solution to $(m|E)$ is represented by a solution to $(m|\sigma(E))$.

We define the notion of *span* to clarify when exactly a TCOB preserves a solution:

Definition 3.2. *Let σ be a trapezoidal change of basis defined by $\sigma(x_i) = E_i$ for each $i = 1, \ldots, n$. Let $\vec{v}$ and $\vec{\eta}$ be vectors. We say $\vec{v} = \sigma[\vec{\eta}]$ if and only if $\vec{v} = (E_1[\vec{\eta}], \ldots, E_n[\vec{\eta}])$. We define* $\text{span}(\sigma) = \{\vec{v} : \exists\vec{\eta}.\, \vec{v} = \sigma[\vec{\eta}]\}$.

Lemma 3.3. *For every $E \in \mathscr{E}$ we have $E[\sigma[\vec{\eta}]] = \sigma(E)[\vec{\eta}]$.*

Proof. By induction on the structure of E. □

Lemma 3.4. *Let $(m|E)$ be a solvable divisibility constraint. Then there exists a trapezoidal change of basis σ such that*

1. $\sigma(E) = mE'$ *for some* E', *and*
2. $\forall\vec{v}.\ (m|E[\vec{v}]) \Longrightarrow \vec{v} \in \text{span}(\sigma)$.

Note that first condition implies that every vector is a solution to $(m|\sigma(E))$.

Proof. Induction on n, the number of dimensions. In the base case $n = 0$ for which case $E = c$ where $c \in \mathbb{Z}$. Take σ to be the identity map on $\mathscr{E}$. Since $(m|c)$ is solvable we have $c = md$ for some $d \in \mathbb{Z}$. Thus $\sigma(E) = md$, satisfying property 1. The second property is trivially true since there is only a single (empty) vector of dimension 0.

Now consider the case for $n > 0$ dimensions. Let $E = F + cx$ where $dim(F) < n$, $c \in \mathbb{Z}$, and $dim(x) = n$. Let $g = \gcd(c, m)$ and let c' and m' be such that $c = gc'$ and $m = gm'$. Since $(m|E)$ is solvable, $(g|E)$ must also be solvable. Furthermore, we know $(g|c)$ so $(g|F)$ is solvable. By the inductive hypothesis, let σ' be a TCOB with

1. $\sigma'(F) = gF'$ for some F', and
2. $\forall\vec{v}.\ (g|F[\vec{v}]) \Longrightarrow \vec{v} \in \text{span}(\sigma')$.

Then, let σ be an extension of σ' with $\sigma(x) = m'x - \text{invmod}(c', m')F'$. Here $\text{invmod}(c', m')$ is the multiplicative inverse of c' modulo m' which is well-defined since $\gcd(c', m') = 1$. Note also that

$dim(\sigma(x)) = dim(x)$. Using σ we have,

$$\begin{aligned}
\sigma(E) &= gF' + c\sigma(x) \\
&= gF' + c(m'x - \text{invmod}(c', m')F') \\
&= cm'x + gF' - c\,\text{invmod}(c', m')F' \\
&= cm'x + gF' - gc'\,\text{invmod}(c', m')F' \\
&= cm'x + g(1 - c'\,\text{invmod}(c', m'))F' \\
&= cm'x + g(1 - (1 + m'k))F' && \text{for some } k \in \mathbb{Z} \\
&= cm'x + gm'kF' \\
&= gc'm'x + gm'kF' \\
&= gm'(c'x + kF') \\
&= m(c'x + kF')
\end{aligned}$$

So that σ satisfies property 1.

To prove property 2 on σ, let $\vec{v}$ be a vector such that $(m|E[\vec{v}])$. Then we have $(m|(F[\vec{v}] + c\vec{v}_x))$ where $\vec{v}_x$ is the x-component of $\vec{v}$. This implies $(g|F[\vec{v}])$ and so by the inductive hypothesis we have $\vec{v}' \in span(\sigma')$ where $\vec{v}'$ is $\vec{v}$ without the final x-component. This means there is a vector $\vec{\eta}'$ such that $\vec{v}' = \sigma[\vec{\eta}']$. To show $\vec{v} \in \text{span}(\sigma)$ we want to extend $\vec{\eta}'$ with an x-component, $\vec{\eta}_x$, such that $\vec{v} = \sigma[\vec{\eta}]$. Such a $\vec{\eta}_x$ exist if and only if we can satisfy $\vec{v}_x = m'\vec{\eta}_x - \text{invmod}(c', m')F'[\vec{\eta}']$. This equation will have an integral solution for $\vec{\eta}_x$ if and only if

$$\begin{aligned}
\vec{v}_x &\equiv -\text{invmod}(c', m')F'[\vec{\eta}'] \pmod{m'} && \iff \\
\vec{v}_x + \text{invmod}(c', m')F'[\vec{\eta}'] &\equiv 0 \pmod{m'} && \iff \\
c'\vec{v}_x + c'\,\text{invmod}(c', m')F'[\vec{\eta}'] &\equiv 0 \pmod{m'} && \iff \\
c'\vec{v}_x + F'[\vec{\eta}'] &\equiv 0 \pmod{m'} && \iff \\
gc'\vec{v}_x + gF'[\vec{\eta}'] &\equiv 0 \pmod{gm'} && \iff \\
c\vec{v}_x + \sigma(F)[\vec{\eta}'] &\equiv 0 \pmod{m} && \iff \\
c\vec{v}_x + F[\vec{v}'] &\equiv 0 \pmod{m}
\end{aligned}$$

Note that $F[\vec{v}'] = F[\vec{v}]$ since $dim(F) < n$. Thus we only need to show $(m|(c\vec{v}_x + F[\vec{v}]))$ which is equivalent to $(m|E[\vec{v}])$. This is true by assumption. Thus property 2 also holds on σ. $\square$

The proof of Lemma 3.4 contains an algorithm for recursively computing a TCOB σ for any solvable divisibility constraint $(m|E)$. We demonstrate this algorithm in the following example.

Example 3.5. *Consider the divisibility constraint* $(2|(1+x+y))$ *where* $dim(x) = 1$ *and* $dim(y) = 2$. *The algorithm implicit in the proof of Lemma 3.4 produces the TCOB* $\sigma(x) = x$, $\sigma(y) = 2y - x - 1$. *Letting* $E = 1 + x + y$ *we have* $\sigma(E) = 2y$ *so that* $(2|\sigma(E))$ *is true for all vectors.*

Given a constraint $x_n = (N_k * x_k + \ldots + N_0)/D_n$ we have shown that we can compute a TCOB σ so that $(D_n|(N_k * x_k + \ldots + N_0))$ is always satisfied, allowing us to always compute an integral value for x_n. We can translate any such satisfying solution $(\vec{\eta})$ into a solution of our original trapezoid via $\vec{v} = \sigma[\vec{\eta}]$. It is easy to show that σ preserves all constraints on trapezoids. Therefore, we can repeatably apply this procedure starting from the highest dimensions and working our way down, eliminating all non-trivial integral constraints along the way. Our final step is to update our reference vector to reflect the change of base, $\vec{v}' = \sigma^{-1}[\vec{v}] = \{\vec{w} : \vec{v} = \sigma[\vec{w}]\}$. We know that such a vector exists because the span of σ includes all of the vectors that satisfied the original constraint, including the original reference vector.

$$\frac{\begin{array}{l} I_n \equiv (x_n = P/D_n) \\ x_n \in \mathbb{Z} \quad (D_n|P) \to \sigma_n \quad I'_n = (x_n = \sigma_n(P)/D_n) \\ \sigma' = \sigma_n(\sigma) \quad \vec{v}' = \sigma_n^{-1}(\vec{v}) \quad T'_m = \sigma_n(T_m) \end{array}}{I_n \cap T_m,\, \sigma,\, \vec{v} \xrightarrow{IE} I'_n \cap T'_m,\, \sigma',\, \vec{v}'} \text{ INTEGER-EQUALITY}$$

3.4 Interval Restriction

While double bounded intervals are known to be satisfiable at the reference vector, there is no guarantee that they are satisfiable under arbitrary variable assignments. To ensure satisfiability, we need to ensure that, for any valid variable assignment, the upper bound will not be less than the lower bound and, for integer variables, that an integer solution exists between the upper and lower bound.

For all rational variables and for integer variables when $lcd(P^L) = 1$ or $lcd(P^U) = 1$ we simply intersect the domain restriction $\langle\langle P^L \leq P^U \rangle\rangle$ with the remaining trapezoid. This ensures that, for any variable assignment that satisfies the newly restricted trapezoid, the lower bound will not be greater than the upper bound. For integer variables this restriction is sufficient because either P^L or P^U will be an integer valued polynomial and, as a result, when $P^L \leq P^U$ is true it always contains an integer solution.

$$\frac{\begin{array}{l} I_n = (P^L \leq x_n) \cap (x_n \leq P^U) \\ (\, x_n \in \mathbb{Q} \vee lcd(P^L) = 1 \vee lcd(P^U) = 1 \,) \\ \langle\langle P^L \leq P^U \rangle\rangle \cap T_m \overset{\vec{v}}{\longrightarrow}{}^{*} T'_m \end{array}}{I_n \cap T_m,\, \sigma,\, \vec{v} \xrightarrow{IR} I_n \cap T'_m,\, \sigma,\, \vec{v}} \text{ INTERVAL-RESTRICTION-1}$$

When $lcd(P^L) > 1$ and $lcd(P^U) > 1$ the constraint $P^L \leq P^U$ is not sufficient to ensure an integer solution for x because P^L and P^U are not guaranteed to be integer valued for every variable assignment. The constraint $P^L + 1 \leq P^U$, on the other hand, does always contain an integer solution. If this constraint contains the reference vector (ie: $P^L[\vec{v}] + 1 \leq P^U[\vec{v}]$) then it is conservative to use this constraint as a domain restriction.

$$\frac{\begin{array}{l} I_n = (P^L \leq x_n) \cap (x_n \leq P^U) \\ x_n \in \mathbb{Z} \\ lcd(P^L) > 1 \\ lcd(P^U) > 1 \\ P^L[\vec{v}] + 1 \leq P^U[\vec{v}] \\ \langle\langle P^L + 1 \leq P^U \rangle\rangle \cap T_m \overset{\vec{v}}{\longrightarrow}{}^{*} T'_m \end{array}}{I_n \cap T_m,\, \sigma,\, \vec{v} \xrightarrow{IR} I_n \cap T'_m,\, \sigma,\, \vec{v}} \text{ INTERVAL-RESTRICTION-2}$$

If $lcd(P^L) > 1$, $lcd(P^U) > 1$, and $P^L[\vec{v}] + 1 > P^U[\vec{v}]$ we perform a change of base to ensure that either the upper or lower bound is always integral. Here we consider the change of base for the lower bound with the understanding that the case of the upper bound is symmetrical. We first adjust (tighten) the lower bound in such a way that it is equal to $x[\vec{v}]$ when evaluated at the reference vector. Note that, because $P^L[\vec{v}]+1 > P^U[\vec{v}]$, there is only 1 integer solution in this interval and $x_n[\vec{v}]$ is it. We compute a change of base for P'^L to ensure that it is always integral. We then restrict the domain of T_m to ensure that the resulting upper bound is never less than our new lower bound.

$$\dfrac{\begin{array}{l} I_n = (P^L \leq x_n) \cap (x_n \leq P^U) \\ x_n \in \mathbb{Z} \quad lcd(P^L) > 1 \quad lcd(P^U) > 1 \\ P'^L = P^L + (x_n[\vec{v}] - P^L[\vec{v}]) \\ D^L = lcd(P'^L) \\ (D^L | P'^L) \rightarrow \sigma_n \\ P'^U = \sigma_n(P^U) \quad P''^L = \sigma_n(P'^L)/D^L \\ I'_n = (P''^L \leq x_n) \cap (x_n \leq P'^U) \\ \sigma' = \sigma_n(\sigma) \quad \vec{v}' = \sigma_n^{-1}(\vec{v}) \quad T'_m = \sigma_n(T_m) \\ \langle\langle P''^L \leq P'^U \rangle\rangle \cap T'_m \xrightarrow{v'}{}^{*} T''_m \end{array}}{I_n \cap T_m, \sigma, \vec{v} \xrightarrow{IR} I'_n \cap T''_m, \sigma', \vec{v}'} \text{ INTERVAL-RESTRICTION-3}$$

4 Implementation and ACL2 Formalization

FuzzM[8] is a model-based fuzzing framework, implemented primarily in Java, that employs Lustre[10] as a modeling and specification language and leverages counterexamples produced by the the JKind[4] model checker to drive the fuzzing process. The FuzzM infrastructure includes support for trapezoidal generalization of JKind counterexamples relative to Lustre models involving linear constraints, integer division and remainder, and uninterpreted functions. Our implementation of trapezoidal generalization requires approximately 5K lines of Java code.

Early versions of the fuzzing framework leveraged an interval generalization capability provided by JKind. At the time we observed that, while the results of interval generalization are trivial to sample, it does not generalize linear model features well. Trapezoidal generalization was envisioned as an extension of interval generalization that would preserve the property of efficient sampling while improving support for linear constraints. In architecting the generalization procedure, however, it soon became clear that we didn't know what it meant for our generalizer to be correct. To address this issue we developed a formalization of generalization correctness in ACL2. In a rump session of the 2017 ACL2 workshop we presented our formalization of generalization correctness, a high-level specification for our implementation, and a proof that our implementation was correct[7]. We also reported that, in proving that our implementation was correct, we had identified an error in our initial high-level specification. Our inability to correctly specify even the high-level behavior of our system without mechanical assistance further motivated us to formalize and verify the correctness of our low-level implementation as well. Much of our low-level ACL2 specification has been transliterated from the Java implementation to ensure that it captures our essential design decisions. While the proofs of low-level correctness were useful in ensuring the absence of off-by-one errors and the like, perhaps the primary benefit of this activity was in the formalization process itself. Just as formalizing our high-level algorithms helped to clarify our software architecture, formalizing the low-level behavior of our algorithms has helped us both to clarify and to simplify the resulting implementation.

We now have a proof that the low-level implementation of our trapezoidal generalization procedure over linear constraints satisfies our notion of generalization correctness. At the heart of our formalization is a library for manipulating, reasoning about, and evaluating linear relations over rational polynomials. On top of this library we introduce a notion of normalized variable bounds upon which we define the trapezoidal data structure and a predicate that recognizes ordered trapezoids and regions. The proof that our low-level implementation satisfies our high level specification was largely an exercise in proving that each low-level function preserved our set of correctness invariants. While the majority of this proof effort was routine, two specific challenges were addressed with specialized ACL2 libraries.

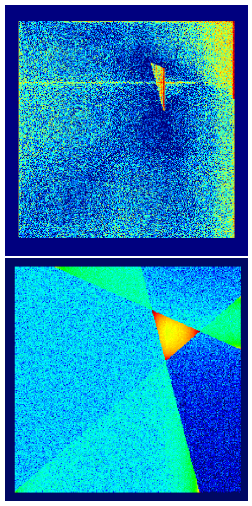
First, to prove that functions operating on normalized variable bounds and trapezoidal regions preserve our variable ordering invariants we employ an arcane feature of the *nary*[5] library that, effectively, enables us to treat subset relations as equivalence relations in appropriate contexts, rewriting subset terms into their super-sets. This feature allows us to specify function contracts using rewrite rules, rather than back-chaining rules, to establish ordering relations among the variables appearing in the function's results. In the supporting materials see `set-upper-bound-equiv` and `set-upper-bound-ctx` defined by the `defequiv+` event in the file `poly.lisp` along with the congruence rule `set-upper-bound--append` and the family of driver rules exemplified by `>-all-upper-bound-backchaining`. See also the nary equivalence relation `set-upper-bound-equiv` as used in `set-upper-bound-equiv-all--bound-list-variables-restrict` to specify a subset rewrite for specific variables returned by the function `intersect` in the file `intersect.lisp` and compare it with the more traditional `subset-p` rule found just above in the lemma `trapezoid-p-restrict`.

Second, because the termination of the function for performing top-level trapezoidal intersection is a bit tricky, in that it is doubly recursive, reflexive, and it relies on the variable ordering property of trapezoids, we introduced its definition using the *def::ung*[9, 6] macro. This allowed us to postpone its termination proof until we had established that the intersection function preserved the variable ordering. We then used *def::total* to prove that the function does, in fact, terminate on well-formed trapezoids. These events are found in the file `intersect.lisp` in the supporting material.

In future work we plan to verify that our method for trapezoidal restriction, as described in Section 3, is correct. Specifically we hope to show that restricted trapezoids are valid generalizations and that they can be sampled to produce satisfying variable assignments without backtracking, even for integer variables. We also hope to extend our infrastructure to allow us to verify that our approaches for generalizing uninterpreted function instances and integer division and remainder, which we do not discuss in this paper, are sound.

5 Related Work

We have compared the performance of our trapezoidal generalization technique with the interval generalization capability provided by the JKind model checker[4]. In all but the smallest models, trapezoidal generalization was competitive with interval generalization in terms of generalization speed. For large models, interval generalizations are actually more expensive to compute than trapezoidal generalizations. This is likely because trapezoidal generalizations are computed in a single symbolic simulation of a model whereas the interval generalization technique requires multiple simulations to compute. Within our fuzzing framework the sampling rates for trapezoidal generalizations are also competitive with interval generalization. Tests in our framework suggest that interval sampling rarely provides better than twice the overall performance of trapezoidal sampling, and is typically less than 50 percent faster[3]. In terms of absolute performance, our deployed system routinely exhibits effective trapezoidal sampling rates on the order of 100's of thousands of generalized variables per second.



Trapezoidal generalization also clearly outperforms interval generalization in terms of preserving linear model features. On the right are two heat maps generated by capturing thousands of samples from generalized solutions to a number of properties involving arbitrary Boolean combinations of three linear constraints. The upper heat map employed interval generalization and the lower trapezoidal generalization. The three linear constraints involved in the various properties are clearly visible in the lower heat map, whereas the upper heat map is dominated by interval generalization artifacts.

Welp[12] discusses a polytope generalization which is more expressive than trapezoidal generalization but is also more expensive to manipulate. Operations on polytopes can be exponential in both time and space. The worst case size of our representation is quadratic in the number of variables. The intersection operation (on ordered trapezoids) is worst case cubic and a naive implementation of post-processing is worst case quartic for integer domains. Sampling a polytope is at least as hard as linear programming and for integers it is NP-complete. After post-processing, sampling a trapezoidal generalization requires only a quadratic number of evaluations. Trapezoidal generalization is thus more expressive than interval generalization but more efficient than polytope generalization, both from a computational point of view as well as from the perspective of sampling.

Symbolic generalization techniques are also used in abstract interpretation[2]. Because efficiency is an overriding concern the abstractions commonly used therein are often quite limited[3]. While more complex and expressive representations are typically less efficient, the octagon domain[11] is actually quite comparable to trapezoids in terms of both computational and representational complexity. Efficient sampling of such abstract domains, however, is generally not a concern. Also, the objective of generalization in abstract interpretation is typically verification; generalization for this purpose is common[1]. When used for verification, generalization should provide a conservative over-approximations of variable domains. Our specification, on the other hand, calls for a conservative under-approximation of such domains. As a result, a generalization technique designed for one domain would be unsound in the other.

6 Conclusion

We have presented a generalization technique for logical formulae described in terms of Boolean combinations of linear constraints that is optimized for use with model-based fuzzing. The generalization technique employs trapezoidal solution sets that can be computed with reasonable space and time bounds and then sampled efficiently. We developed a formal specification of generalization correctness and used it to verify the correctness of key aspects of our generalization algorithm using ACL2. Finally we described a post-processing procedure that produces restricted trapezoidal solutions that can be rapidly and efficiently sampled without backtracking, even for integer domains. The formal verification of this post-processing step is expected to be addressed in future work.

[3]Measurements are based on overall test vector generation rates, not generalization sampling rates in isolation

References

[1] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu & Helmut Veith (2003): *Counterexample-guided Abstraction Refinement for Symbolic Model Checking*. *J. ACM* 50(5), pp. 752–794. Available at `http://doi.acm.org/10.1145/876638.876643`.

[2] Patrick Cousot & Radhia Cousot (1977): *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*. In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, ACM, New York, NY, USA, pp. 238–252. Available at `http://doi.acm.org/10.1145/512950.512973`.

[3] Patrick Cousot & Radhia Cousot (1977): *Static Determination of Dynamic Properties of Generalized Type Unions*. *SIGPLAN Not.* 12(3), pp. 77–94. Available at `http://doi.acm.org/10.1145/390017.808314`.

[4] Andrew Gacek, John Backes, Mike Whalen, Lucas G. Wagner & Elaheh Ghassabani (2018): *The JKind Model Checker*. In: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, pp. 20–27. Available at `https://doi.org/10.1007/978-3-319-96142-2_3`.

[5] 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. Available at `http://doi.acm.org/10.1145/1217975.1217981`.

[6] David Greve (2009): *Assuming Termination*. In: *Proceedings of the Eighth International Workshop on the ACL2 Theorem Prover and Its Applications*, ACL2 '09, ACM, New York, NY, USA, pp. 114–122. Available at `http://doi.acm.org/10.1145/1637837.1637856`.

[7] David Greve (2017): *Generalization Correctness*. `http://www.cs.utexas.edu/users/moore/acl2/workshop-2017/slides-rump/greve-ACL2-2017.pdf`.

[8] David Greve (2018): *FuzzM: Fuzzing with Models*. `https://github.com/collins-research/FuzzM`.

[9] David Greve & Konrad Slind (2013): *A Step-Indexing Approach to Partial Functions*. In Ruben Gamboa & Jared Davis, editors: Proceedings International Workshop on the *ACL2 Theorem Prover and its Applications*, Laramie, Wyoming, USA , May 30-31, 2013, *Electronic Proceedings in Theoretical Computer Science* 114, Open Publishing Association, pp. 42–53, doi:10.4204/EPTCS.114.4.

[10] N. Halbwachs, P. Caspi, P. Raymond & D. Pilaud (1991): *The synchronous data flow programming language LUSTRE*. *Proceedings of the IEEE* 79(9), pp. 1305–1320, doi:10.1109/5.97300.

[11] Antoine Miné (2006): *The Octagon Abstract Domain*. *Higher Order Symbol. Comput.* 19(1), pp. 31–100. Available at `http://dx.doi.org/10.1007/s10990-006-8609-1`.

[12] Tobias Welp & Andreas Kuehlmann (2013): *QF_BV Model Checking with Property Directed Reachability*. In: *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, EDA Consortium, San Jose, CA, USA, pp. 791–796. Available at `http://dl.acm.org/citation.cfm?id=2485288.2485480`.

Incremental SAT Library Integration Using Abstract Stobjs

Sol Swords
Centaur Technology, Inc.
`sswords@centtech.com`

We describe an effort to soundly use off-the-shelf incremental SAT solvers within ACL2 by modeling the behavior of a SAT solver library as an abstract stobj. The interface allows ACL2 programs to use incremental SAT solvers, and the abstract stobj model allows us to reason about the behavior of an incremental SAT library so as to show that algorithms implemented using it are correct, as long as the library is bug-free.

1 Introduction

ACL2 users have long recognized the potential utility of integrating external proof tools [7]. While many such tools have been successfully integrated and used in ACL2 proofs, as far as we know these have all been used in a stateless manner: that is, a complete query is built up within ACL2 and exported in a format accessible to the external tool, then the external tool is executed and finishes, at which point ACL2 reads its output. However, some external tools benefit from storing state between queries. Incremental satisfiability (SAT) solvers, in particular, keep learned facts and heuristic information from previous queries and use these to speed up later queries, allowing for repeated SAT checks that can be much faster than they would be if the solver was started from scratch for every query.

This paper describes an interface allowing ACL2 programs to use an external incremental SAT library in a stateful manner. The library is accessed through an abstract stobj. The abstract stobj interface functions' logical definitions model the behavior of the incremental SAT library, and their executable definitions call into the library to set up queries and get their results. We show how our model of an incremental SAT solver can be used in a complete algorithm, namely *fraiging* or AIG SAT sweeping [8], and the algorithm proved correct assuming correct behavior of the incremental solver.

We begin in Section 3 by discussing the incremental SAT library interface we are targeting. Next we give an overview of the integration in Section 4, describe the logical model in Section 5, and describe the mechanics of interfacing with the external library in Section 6. We assess the soundness of the incremental SAT integration in Section 7. We then describe the implementation of a fraiging algorithm using incremental SAT in Section 8.

2 Related Work

Several other efforts have resulted in integrations between ACL2 and external proof tools. Our work is most directly based on SATLINK [5], which calls an external SAT solver executable on a single problem in a stateless manner. SATLINK provides a function that calls an external SAT solver on a CNF formula; that function is assumed to only return `:unsat` when the formula is unsatisfiable, and this can be used to perform ACL2 proofs using GL [12] or by otherwise appealing to that assumption. Similarly, SMTLINK [9] provides a trusted clause processor which encodes ACL2 formulas as SMT problems and calls the

Z3 SMT solver to prove them, also statelessly. Reeber's SAT-based decision procedure for a decidable subset of ACL2 formulas [10] also calls an external SAT solver statelessly.

Somewhat different in flavor is ACL2SIX [11], which calls IBM's internal SixthSense checker to verify hardware properties. SixthSense, in this case, provides not just the decision procedure but the semantics of the model as well. The ACL2 logical interface to the hardware model consists of two functions `sigbit` and `sigvec`. These functions each take as inputs an *entity* representing a machine and environment model known to the external tool, a signal name, and a time, and they return the value of the signal at the given time. These functions are not defined and cannot be executed, but facts about them can be obtained by calling the ACL2SIX clause processor. This clause processor renders the formula into a VHDL property and calls SixthSense to prove the property. For an adder module, for example, the `sum` signal at time n can be proven to be the bit-vector sum of the `a` and `b` inputs at time $n-1$. Calls into SixthSense are stateless, but because of the simplicity of the logical connection between the external solver and ACL2, a stateful integration could have the same logical story. That is, integration with a SixthSense shared library which collected information about a hardware model across multiple queries could be used with the same logical model.

3 Incremental SAT Interface

Rather than targeting one particular incremental SAT solver, we chose to interface with IPASIR, a simple C API introduced for use in SAT Race 2015 [2] and used through the 2017 SAT competition [3]. (IPASIR stands for Reentrant Incremental SAT solver API, in reverse.) The IPASIR interface consists of the following 10 functions. The API describes the states of the solver object as `INPUT`, `SAT`, or `UNSAT`. Most functions may be used in any of these states, but a few, as noted below, require the solver to be in a particular state.

- `ipasir_signature` returns a name and version string for the solver library.
- `ipasir_init` constructs a new solver object in the `INPUT` state and returns a pointer to it.
- `ipasir_release` destroys a solver object.
- `ipasir_add` adds a literal to the new clause currently being built or, if the input is 0 (which is not a literal), adds that clause to the formula; the resulting solver is in the `INPUT` state.
- `ipasir_assume` adds a literal to be assumed true during the next SAT query and puts the solver in the `INPUT` state.
- `ipasir_solve` solves the formula under the current assumptions, determining whether it is satisfiable or unsatisfiable unless it is interrupted by the `ipasir_set_terminate` callback. Puts the solver into the `INPUT` state if the check failed due to a termination condition, `SAT` or `UNSAT` respectively if satisfiable or unsatisfiable.
- `ipasir_val` returns the truth value of a variable in the satisfying assignment produced by the previous call of `ipasir_solve`; it requires that the solver is in the `SAT` state and leaves it in that state.
- `ipasir_failed` checks whether a given assumption literal was used in proving the previous unsatisfiability result produced by `ipasir_solve`; it requires that the solver is in the `UNSAT` state and leaves it in that state.

- `ipasir_set_terminate` sets up a callback function that will be called periodically by the solver during `ipasir_solve` and can decide whether to interrupt the search. Preserves the current state of the solver.
- `ipasir_set_learn` sets up a callback function that will be called each time a clause is learned, allowing these clauses to be recorded. Preserves the current state of the solver.

The IPASIR interface supports the following basic usage of an incremental solver. The client first creates a solver object using `ipasir_init`, then builds up a formula using repeated calls of `ipasir_-add`. Usually the formula itself should be satisfiable: clauses can only be added and not removed, so once it is determined that the formula is unsatisfiable, no further information can be obtained from subsequent queries—it will always remain unsatisfiable. Instead, the formula is kept satisfiable, but a separate set of assumptions may be provided using `ipasir_assume` that may or may not be satisfiable in conjunction with the formula. A call to `ipasir_solve` checks whether the formula and all the assumptions can be simultaneously satisfied, after which the assumptions are deleted. When the call of `ipasir_solve` returns satisfiable, the satisfying assignment may be queried using `ipasir_val`, and when unsatisfiable, the unsatisfiable subset of the assumptions may be queried using `ipasir_failed`. After the client is done with these queries, it can then add more clauses and/or assumptions before solving again. (Once the client starts adding clauses or assumptions, it is no longer allowed to query the satisfying assignment or unsatisfiable subset until after the next call to `ipasir_solve`.) When done, the client calls `ipasir_-release` to free the solver's memory.

Our ACL2 interface supports this usage pattern, but it does not yet support `ipasir_set_learn` and only supports `ipasir_set_terminate` in a limited fashion, allowing searches to be interrupted after the callback is called some number of times. Some of the other functions are modified so as to make their output more idiomatic in ACL2. For example, the C interface for `ipasir_solve` returns 10 when satisfiable, 20 when unsatisfiable, and 0 when interrupted; we instead return one of the symbols `:sat`, `:unsat`, or `:failed`.

4 Overview of ACL2 Integration

The logical model of the IPASIR interface is contained in a book named `ipasir-logic.lisp`, separate from the code which interfaces with the external library. This book is purely in the ACL2 logic, and its only trust tag is used to flag a function as having an unknown constraint, which in itself cannot cause unsoundness. Books defining programs that use IPASIR functionality can therefore remain free of external code or under-the-hood hacks. They will simply require that the book defining the executable backend interface, `ipasir-backend.lisp`, be loaded before any IPASIR functions can be actually executed.

The `ipasir-logic` book defines an abstract stobj that provides the interface to the incremental SAT library. Abstract stobjs were conceived as a mechanism to replace a complicated executable implementation with a simpler logical model that still accurately represents the functionality of that implementation [6]. Our use of abstract stobjs for interfacing with an external library is in this same spirit, differing in that the executable definitions used in admitting the abstract stobj are later replaced (when `ipasir-backend` is loaded) by under-the-hood implementations that use the external library.

The features of abstract stobjs are a very good fit for the requirements of interfacing with an external library:

- Usage of the library is forced to follow the stobj discipline, i.e., any operation that changes the state of the external solver object must return that object, and that return value must replace the (single) existing binding of that object.
- The set of interface functions can be restricted to a small API. This allows the logical model to contain data that can't be accessed in executable code (and therefore need not be computed) but may still be important for modeling the behavior.
- The interface functions' guards may be engineered to prevent ill-defined behaviors and illegal interactions.

5 Logical Model

Our logical model of an incremental SAT solver is built on the existing theory of conjunctive normal form satisfiability provided by the SATLINK library [5]. In particular, a literal is represented as a natural number with its least significant bit representing its polarity and the rest of its bits giving the index of its variable; a clause is a list of literals, and a formula is a list of clauses. The representation of literals is a trivial difference between the ACL2 interface and the C interface, which uses signed non-zero integers with negation giving the sign and the absolute value giving the variable number. We will refer to these C-style literals as DIMACS literals, as opposed to SATLINK literals.

The IPASIR solver is accessed via an abstract stobj called `ipasir`. The logical model of that abstract stobj is an object of type `ipasir$a`, which is a product type containing the following fields:

- `formula`, a list of lists of literals, representing the permanent formula stored in the solver
- `assumption`, a list of literals, the current assumption
- `new-clause`, a list of literals, the current clause being built
- `status`, one of the symbols `:undef`, `:input`, `:unsat`, or `:sat`, the current state of the solver
- `solution`, a list of literals, either a satisfying assignment or an unsatisfiable subset of the previous assumption
- `solved-assumption`, a list of literals, the previous assumption that was proved unsatisfiable (used in the guard for `ipasir-failed`, described below)
- `callback-count`, the number of times the callback function (set in the underlying solver with `ipasir_set_terminate`) has been called during solve
- `history`, a list representing the history of all operations performed on the solver object.

The abstract stobj interface only allows direct access to a few of these fields. The others are logical fictions which are convenient for modeling the state of the underlying solver implementation but are never actually built. We describe the full abstract stobj interface below, but we begin by briefly describing how the history field is updated.

The history field is a record of all updates performed on the solver object. Each updater operation adds an entry to the history, which ensures that the solver object is never equal to a previous version of itself. Additionally, before the solver object can be used the history must be initialized with an object read from the ACL2 state's oracle, which prevents any two solvers from being provably equal. This removes a source of unsoundness due to nondeterminism, discussed in more detail in Section 7.2. In our description of the operations below, we will omit discussion of the history field because it is always updated by simply consing on a new entry.

We now describe how a solver object is initialized and released. When it is created, an `ipasir` stobj initially has status `:undef` and an empty history. The only interface function that allows us to progress from this state is `ipasir-init`, whose guard only requires that the status be `:undef`. `ipasir-init` initializes all non-history fields to default values except it sets status to `:input`. It uses the ACL2 state to add to the history a value read from the state's oracle field, and returns a modified state with that value removed from the oracle. The solver object is then usable, having status `:input`. When done, it should be freed using `ipasir-release`, which sets the status field back to `:undef`. (If a `with-local-stobj` form creating an `ipasir` object is exited without calling `ipasir-release`, the memory used by the backend solver object will be leaked.) The solver can be reinitialized after releasing it using `ipasir-reinit`, which is much like `ipasir-init`, but does not take state and cannot be used for the first initialization (it requires that the history field is non-empty).

The remaining functions support the basic usage model of an incremental solver. The following functions are used to set up the problem to be solved; they may be used in any initialized state (i.e., their guards only require that status is not `:undef`), and all set the status to `:input`:

- `(ipasir-add-lit ipasir lit)` conses the given literal onto the `new-clause` field.
- `(ipasir-finalize-clause ipasir)` adds the current `new-clause` to the formula and empties it.
- `(ipasir-assume ipasir lit)` conses the given literal onto the `assumption` field.
- `(ipasir-input ipasir)` only sets the status to :input.

The `ipasir-input` is convenient when defining functions that may add some assumptions or clauses but sometimes do nothing; in this case, if one calls `ipasir-input` instead of doing nothing, then the status of the resulting solver will always be `:input`. This is allowable since any interface function that can be called in the `:input` state may be called in any state other than `:undef`.

After setting up the problem, `(ipasir-solve ipasir)` is used to check satisfiability. This is a constrained function which returns a search status as well as a new solver object. Its guard requires that status is not `:undef` and that the `new-clause` is empty. (This requirement simply removes an ambiguity from the interface specification: it isn't clear what it means if some literals have been added to a new clause but the clause has not been finalized when the solver is called.) The constraints require:

- The search status returned will be `:failed`, `:unsat`, or `:sat`.
- The resulting solver will have status `:input` if failed, `:unsat` or `:sat` correspondingly otherwise.
- If `:unsat`, the solution field of the result solver contains a subset of the input solver's assumption, and that subset cannot be satisfied in conjunction with the formula.
- If `:unsat`, the solved-assumption field of the new solver equals the assumption field of the input solver.
- The assumption and new-clause fields of the resulting solver are empty, and the formula is preserved from the input solver.
- The callback count of the resulting solver is greater than or equal to that of the input solver.

We could assume that the solver produces a satisfying assignment when it returns `:sat`, but in most applications it is easy to check that the assignment is correct, if necessary.

The constraints of the `ipasir-solve` function do not fully determine its behavior—for example, it is allowed to return `:failed` on any input, even when it could alternatively return `:sat` or `:unsat`. When actually executed with a backend solver loaded, `ipasir-solve` will return the answer supplied by the solver—this gives us access to facts about `ipasir-solve` that are not implied by its constraints. In ACL2's usual treatment of constrained functions, the constraints are assumed to be everything that is known about the function *a priori*, and to imply everything that is later proved about it. Any facts proved about a constrained function can then be *functionally instantiated*, i.e., assumed true of any other function that satisfies the constraints [4]. Since this is not true of the constraints for `ipasir-solve`, we say that it has *unknown constraints*[1] [7], which prevents functional instantiation of facts we have proved about it.

After solving, if the result was `:sat` or `:unsat`, the solver may be queried to derive the satisfying assignment or the unsatisfiable subset of the assumptions, respectively:

- `(ipasir-val ipasir lit)` requires that status is `:sat` and returns 1, 0, or `NIL` depending whether that literal is true, false, or undefined in the satisfying assignment (i.e. the solution field).
- `(ipasir-failed ipasir lit)` requires that status is `:unsat` and that the literal is a member of the solver's solved-assumption field, and returns 1 if the literal is a member of the identified unsatisfiable subset (i.e. the solution field), 0 if not.

In many algorithms it's desirable to limit the amount of time spent trying to solve any one query. We support this via the function `(ipasir-set-limit ipasir limit)`, where limit is a natural number or `NIL`; passing a natural number here will cause the solver to fail a call of `ipasir-solve` after that many callbacks, and passing `NIL` will remove that limit. Logically, this only affects the history and resets the callback count to 0. The callback count can be accessed for performance monitoring using `(ipasir-callback-count ipasir)`.

Rob Sumners contributed an update to the IPASIR integration that adds to the abstract stobj interface the functions necessary to make all the guards executable. In earlier versions, the guards for the interface functions were all non-executable, which in practice meant that all execution must be done on `ipasir` objects created by `with-local-stobj`, not the global `ipasir` object. An `ipasir` object newly created by `with-local-stobj` is known to be in a certain state, so functions that use this mechanism could have verified, executable guards even if they called ipasir interface functions that have non-executable guards. For example, the guard for `ipasir-init` is:

```
  (non-exec (eq (ipasir$a->status ipasir) :undef))
```

This needed to be non-executable because `ipasir$a->status` is just the logical model, which can't be executed on the stobj, and there was no abstract stobj interface function that could return the status or check whether it was `:undef`. However, a function with verified, executable guards could be created using `with-local-stobj` as follows, because the `ipasir` object created is known to have `:undef` status:

```
(defun ipasir-initialize-and-release (state)
  (declare (xargs :stobjs state))
  (with-local-stobj ipasir
    (mv-let (ipasir state ans)
      (b* (((mv ipasir state)
```

[1] The unknown constraint is currently added using the `define-trusted-clause-processor` utility, but it may be supported more directly by the ACL2 system in the future.

```
          (ipasir-init ipasir state))
         (ipasir (ipasir-release ipasir)))
      (mv ipasir state nil))
    (mv state ans))))
```

Sumners added the following interface functions, which suffice to express all the guards as executable terms:

- `ipasir-get-status` returns the solver status, `:undef`, `:input`, `:unsat`, or `:sat`
- `ipasir-some-history` returns `T` if the history is nonempty
- `ipasir-empty-new-clause` returns `T` if the `new-clause` is empty
- `ipasir-get-assumption` returns the list of current assumption literals
- `ipasir-solved-assumption` returns the assumption before the last solve, if the solver produced `:unsat`.

Sumners additionally added two extra interface functions that require library support that is not part of the IPASIR API, but still can be easily supported by most incremental SAT libraries. If the external library is set up to support these, then an extra book can be loaded which supplies their actual implementation; otherwise, stub functions are used instead. The two functions:

- `ipasir-bump-activity-vars` increases the activity heuristic of the variables of the given literals
- `ipasir-get-curr-stats` returns several counters useful for heuristically monitoring the current size and complexity of the solver's formula.

One remaining function, `(ipasir-signature)`, is supported through another mechanism, in case the user needs to access the solver library's version information. The `ipasir-signature` function is constrained to return a string. When the backend is loaded, we define a new function (in raw Lisp) that returns the result from the external library's `ipasir_signature` call. We use `defattach` to attach this function to `ipasir-signature`. Defattach is designed to allow constrained functions to execute only in contexts where their results can't be recorded as logical truths. Otherwise we could prove via functional instantiation that any function that always returns a string returns the same value as `ipasir-signature`. We still need to trust that the backend library always returns the same string from `ipasir_signature`; otherwise we could prove NIL using, for example, a clause processor that checks `(equal (ipasir-signature) (ipasir-signature))`.

6 Interfacing with the External Library Backend

The interface with the external IPASIR implementation library is defined in the book `ipasir-backend`. This loads (in raw Lisp) an external shared library specified by the environment variable `IPASIR_-SHARED_LIBRARY`, then redefines the executable functions of the abstract stobj interface so that they call into the external library. We use the Common Lisp CFFI (Common Foreign Function Interface) package available through Quicklisp to load and call functions in the shared library.

The real underlying object on which the ipasir interface functions are run is a vector containing the following seven pieces of data:

0. the foreign pointer to the backend solver object used by the C API

1. the foreign pointer to the structure tracking the callback count and limit
2. the current solver status (`:undef`, `:input`, `:unsat`, or `:sat`)
3. a Boolean saying whether the current new clause is empty
4. a Boolean saying whether the current history is nonempty (that is, whether the solver has ever been initialized)
5. a list of literals tracking the current assumption
6. a list of literals which is the previous solved assumption if status is `:unsat`.

To install the backend interface, we redefine all the executable interface functions of the abstract stobj so that they run the appropriate operations on this vector. These executable interface functions are defined in ACL2 as operating on the concrete stobj `ipasir$c` that was used to define the abstract stobj `ipasir`. (We discuss a soundness problem that arises from this redefinition in Section 7.3.)

The backend functions are mostly simple wrappers around calls of the appropriate foreign library functions on the backend solver object (slot 0 of the vector), along with a small amount of bookkeeping to maintain the other fields of the vector. The wrappers also transform input literals from SATLINK to DIMACS format, and translate some query results to make them more idiomatic in Lisp. The initialization functions also handle errors that might occur due to the external library not being loaded, catching the raw Lisp error and producing an ACL2 hard error instead. The `ipasir-set-limit$c` function sets up a callback (created using CFFI's `defcallback`) that counts the number of times it is called and returns 1 to end the SAT search if a limit is reached. To support this, `ipasir-init$c` and `ipasir-reinit$c` reset the count to 0 and the limit to `NIL`, and `ipasir-solve$c` resets the count to 0 before beginning the SAT search.

7 Soundness Assessment

While we can't show our integration to be sound via formal proof —in fact, we know of one soundness bug, discussed in Section 7.3— we hope that it can be accepted through a social process of discussing and eliminating potential problems. Similar to ACL2 itself, we hope that over time it can be inspected and determined to be largely sound. We prioritize soundness problems that might be encountered by accident and yield undetected false results, rather than those which would need to be exploited intentionally.

To start the assessment of the IPASIR integration's soundness, we think it is useful to sort the problems that could cause unsoundness into three main categories. These distinctions are blurry, but useful at least in describing what we think is the state of the integration's soundness or lack thereof. We will describe the three categories and then devote a subsection to each category and our efforts to avoid unsoundness of that kind.

First, we may have soundness problems due to mismatches between the behavior of the external library and our assumptions about that behavior. These might be bugs in the library or simply invalid assumptions on our part. Out of the three categories, we believe these are the most critical, since they are most likely to yield undetected false results.

Second, we may have other logical problems not directly due to invalid assumptions about the behavior of the library. We focus here on the problem of ostensible functions that may actually return different values on the same inputs.

Third, we may have soundness problems due to incidental misfeatures of our integration mechanisms, rather than due to the logical modeling of the external library. We know of one soundness bug in this area that remains unfixed, though it would be implausible for it to be exploited unintentionally.

7.1 Validity of Assumed Behavior

We first discuss potential problems due to mismatches between the assumed and actual behavior of the external incremental SAT library (along with the raw Lisp code interfacing with it). Of course, we first have to assume that the external SAT library is bug-free: if the external solver's `ipasir_solve` routine produces a wrong answer, then our interface is not sound. Beyond this, our main defense against these problems is to carefully assess what we are assuming about each interface function. We model our correctness argument on the ACL2 proof obligations necessary to admit an abstract stobj [6]. That is, it suffices to prove, for some correlation relation between the logical model state and the underlying implementation state:

- The logical model and the underlying implementation of the stobj creator function produce initial values satisfying the correlation.
- For each updater, if its logical model and implementation are passed objects satisfying the correlation and the updater's guard, their respective results satisfy the correlation.
- For each accessor, if its logical model and implementation are passed objects satisfying the correlation and guard, their results are equal.
- For each interface function, the guard of the logical model implies the guard of the implementation.
- The logical model of each updater preserves the well-formedness predicate.

The last two items are trivial in our case: our implementations are in raw Lisp and have no guards, and the well-formedness requirement has nothing to do with the implementation side and thus is already proved when admitting the abstract stobj in the logical model.

The other three requirements depend on the correlation relation that we maintain. We can't state this in the ACL2 logic since it involves the raw Lisp and external C library implementation, but we nevertheless try to describe it precisely:

- The `formula` must be logically equivalent to the set of clauses stored in the solver object, which is field 0 of the implementation vector. (The implementation solver may simplify the formula, so it may not be stored in the same form as in the model.)
- The `assumption` must reflect the set of assumption literals in the solver, and must also equal field 5 of the implementation vector.
- The `new-clause` field must reflect the clause under construction of the solver object, and must additionally be empty if and only if field 3 of the implementation vector is true.
- The `status` must be equal to that recorded in field 2 of the implementation vector, and also correspond to the solver object's current state.
- The `solution`, when in the `:unsat` state, must correspond to the solver's recorded unsatisfiable subset of the assumptions, and when in the `:sat` state, must correspond to the solver's recorded satisfying assignment.
- The `solved-assumption`, when in the `:unsat` state, must equal field 6 of the implementation vector, which must also be the set of assumptions from the last call of `ipasir-solve`.
- The `callback-count` must equal the count of callbacks stored in field 1 of the implementation vector, or 0 of that field is a null pointer.
- The `history` must be nonempty if and only if field 4 of the implementation vector is true.

There are 20 functions (including `create-ipasir` and all accessors/updaters) in the abstract stobj interface. Of these, nine are purely accessors, 10 are purely updaters, and one (`ipasir-solve`) is both. To fully argue the correctness of all of these with respect to the correlation relation above, we'd need to justify the correlations for each of the eight fields for each of the updaters, and additionally argue that the model and implementation of each accessor return equal values when the correlation holds. For most interface functions, this is a straightforward but tedious argument. However, `ipasir-solve` is a special case that requires additional explanation.

Rather than a full definition, `ipasir-solve` has constraints that do not fully specify what its result must be. In fact, its constraints are not sufficient to prove that it preserves the correlation relation. For example, suppose that on some inputs the implementation of `ipasir-solve` produces a satisfiable result and therefore ended in the `SAT` state, but the logical model instead produces `:failed` and therefore ends in the `:input` state. This is consistent with the constraints for `ipasir-solve`, which allow it to return `:failed` for any input. But the Lisp definition of `ipasir-solve` simply translates the result from the implementation library into the ACL2 idiom, so this can't happen—as discussed in Section 5, the logical model of `ipasir-solve` is given by its observed behavior, not by its constraints. We do need the constraints to be consistent with all such concrete executions, which we argue, as with the other interface functions, by appealing to the API description and knowledge of what a SAT solver is supposed to do.

7.2 Logical Consistency

Aside from mismatches between our logical assumptions and implementation realities, other possible soundness problems may occur due to inconsistencies in the logical story. In particular, we'll discuss how we addressed nondeterminism, which might otherwise cause ostensible functions to return different results on the same inputs, leading to unsoundness. We have already discussed the incompleteness of the constraints on `ipasir-solve`, which is another example of such a problem; this was a soundness bug in a previous version of the library, which we solved by adding unknown constraints to `ipasir-solve`.

Nothing in the IPASIR interface specification implies that the solver library must be fully deterministic. We therefore need to expect that the solver may produce different results given the same inputs – i.e. the interfaces to the solver are not actually functions. This could easily lead to unsoundness. Specifically, if we could arrange for some interface function to be called twice on inputs that are provably equal and return different results, we could use this to prove `NIL`. Since we can't control the results returned by the underlying solver, we instead ensure that we can't run a vulnerable interface function twice on provably equal inputs. We therefore seek to prevent:

- *Coincidence*: creation of two distinct `ipasir` objects that are provably equal
- *Recurrence*: recreation of a solver state provably equal to a previous state after changing it in a way that might affect the answers returned from queries.

To prevent *coincidence*, we require that a solver must always be initialized for the first time by `ipasir-init`, which seeds the solver's history field with an object taken from the ACL2 state's oracle field, removing that object from the oracle. The oracle is a mechanism by which ACL2 models nondeterminism; it is simply an object about which nothing is initially known, and which can only be accessed by `read-acl2-oracle`, which returns the oracle's `car` and replaces the oracle with its `cdr` [1].) This ensures that we can't prove anything about what an oracle read will produce until that read happens. The object read from the oracle in this case is written to the solver's history field, which has no accessors in the abstract stobj interface, so we can't determine after the fact what object was read from the oracle, either. Additionally, there is no interface function that removes elements from the history, so that object remains there permanently. Since we can't determine the value stored in that field, and since any two solver objects will be seeded upon initialization with two independent oracle reads, we can't prove them to be equal once they are initialized.

To prevent *recurrence*, every operation that changes the external library's state is modeled as consing some additional object onto the history. There is no operation that clears the history or removes any element from it. Therefore a solver object can never be made to go back to a previous state, because the length of its history always increases.

The abstract stobj interface functions which are just accessors (that is, they don't return a modified `ipasir` object) are assumed to be read-only and not affect the state of the underlying solver; therefore, they don't need to update the history. Of these, `ipasir-get-curr-stats`, `ipasir-val`, and `ipasir-failed` make library calls that might affect the external solver object, but if any of these affected the solver state in an observable way we would view it as a bug in the external library. Additionally, `ipasir-input` doesn't need to update the history because it doesn't touch the external solver object.

7.3 Integration Artifacts

A third class of soundness problems arise from factors that we view as unrelated to the logical story of the IPASIR integration; they are merely artifacts of the ACL2 mechanisms that we used or abused in order to achieve the integration. In particular, the `defabsstobj` event is almost exactly what we need to implement an external library interface like this one. However, this requires us to supply a fake implementation using a concrete stobj, in our case `ipasir$c`. This is perhaps unnecessarily complicated and is fertile ground for unsoundness.

For example, in order to install the implementations of the abstract stobj interface functions, we redefine the executable versions of those functions, which were originally defined as operations on the concrete stobj `ipasir$c`. A soundness problem arises here because users could apply these functions after redefinition to the `ipasir$c` object or any stobj congruent to `ipasir$c`. But the interface to `ipasir$c` is not restricted in the same way as the `ipasir` abstract stobj—in fact, it has several low-level accessors, such as one that purports to retrieve the `ipasir$a` object that logically models the solver's behavior. The use of such an accessor could easily lead to unsoundness. For example, the following function can easily be shown to always return `T` as its first return value, but its execution (after loading the backend) returns `NIL`:

```
(define ipasir$c-contra (state)
  (with-local-stobj ipasir$c
    (mv-let (ans state ipasir$c)
      (b* (((mv ipasir$c state) (ipasir-init$c ipasir$c state))
           (solver (ipasir-get ipasir$c)))
        (mv (ipasir$a-p solver) state ipasir$c))
      (mv ans state))))
```

To prevent this, we make the redefined functions *untouchable*, which disallows users from calling these functions or defining new functions that call them. Unfortunately, this also prevents the creation of new abstract stobjs congruent to `ipasir`, which is often desirable. Also unfortunately, this mitigation doesn't completely solve the problem. A determined user can defeat the untouchability of any function defined outside the ACL2 system as follows: copy all the events needed to admit the function, define a wrapper for that function, then load the book that declares the function untouchable. After that point, simply call the wrapper instead of the untouchable function.

We hope to solve this problem more comprehensively in future work, perhaps by adding some features to the ACL2 system. A relatively easy solution for the specific problem above would be to allow concrete stobjs to be defined with non-executable accessors and updaters. Then the only functions that could be executed on that stobj would be ones that were redefined under the hood—namely, the abstract stobj interface functions. A more heavyweight but perhaps also more direct solution would be to extend ACL2 with support for a `defabsstobj` variant intended for this sort of application, perhaps avoiding the introduction of an underlying concrete stobj altogether.

One other known extralogical problem has to do with ACL2's `save-exec` feature. Under normal circumstances this feature can be used to save an executable memory image of the running ACL2, so that it can be restarted from the current state. However, foreign objects cannot be saved in the heap image. Therefore, if we save an executable in which the global `ipasir` object is initialized, the underlying solver object will not exist when the image is executed. Running any ipasir interface functions then will at minimum cause a raw Lisp error and could potentially cause memory corruption or unsoundness. This problem can be avoided by ensuring that live ipasir stobjs are in the `:undef` state before saving an executable.

8 Application: AIG SAT sweeping

We built on the IPASIR integration to implement SAT sweeping, or fraiging, on top of the AIGNET and-inverter graph (AIG) library [5]. Circuit structures such as AIGs are often a good target for incremental SAT, since the logical relationships among the wires can be encoded in the permanent formula and the various queries encoded in the assumptions. During each SAT check, the solver accumulates learned clauses and heuristic information about the circuit that can be used on subsequent checks.

The purpose of SAT sweeping [8] is to search for and remove redundancies in the AIG; that is, to find pairs of nodes that are provably equivalent and remove one of them, connecting its fanouts to the other. This reduces the size of the graph and speeds up subsequent algorithms while preserving combinational equivalence; that is, the combinational formulas of corresponding outputs or next-states in the input and output networks are equivalent. This is a powerful algorithm for combinational equivalence checking because often the two circuits contain many equivalent internal nodes, and finding these equivalent internal nodes makes it much easier to prove the full circuits equivalent.

As a preliminary requirement for SAT sweeping, we need to be able to use SAT to check equivalences between AIG nodes. We use a standard Tseitin transformation [13] to encode substructures of the AIG into CNF as needed. Whenever we need to do a SAT check involving some node, we encode the fanin cone of that node into the CNF formula. This results in a CNF variable corresponding to that node. This process maintains the invariant that each evaluation of the AIG maps to a satisfying assignment of the CNF formula, where for each AIG node that has a corresponding CNF variable, the assignment to the variable is the same as the value of the node:

```
(forall (invals regvals cnf-vals)
        (equal (satlink::eval-formula
                (ipasir::ipasir$a->formula ipasir)
                (aignet->cnf-vals
                 invals regvals cnf-vals sat-lits aignet))
               1))
```

In the above formula, `sat-lits` is a stobj containing a bidirectional mapping between AIG literals and SAT literals. The `invals` and `regvals` are assignments to the AIG's primary inputs and registers (which are treated as combinational inputs for this purpose). The function `aignet->cnf-vals` maps the evaluation of the AIG given by `invals` and `regvals` to an assignment of the CNF variables, replacing the relevant slots of stobj `cnf-vals`; specifically, it satisfies:

```
(implies (sat-varp m sat-lits)
         (equal (nth m (aignet->cnf-vals
                         invals regvals cnf-vals sat-lits aignet))
                (lit-eval
                 (sat-var->aignet-lit m sat-lits)
                 invals regvals aignet)))
```

That is, each variable in the CNF formula is assigned the evaluation of its corresponding AIG literal, namely `(sat-var->aignet-lit m sat-lits)`. The invariant above says that this is always a satisfying assignment for the CNF formula. Therefore, if we obtain an UNSAT result, it must be that the added assumptions, `(ipasir$a::assumption ipasir)`, are to blame. In particular, an UNSAT result implies that no evaluation of the AIG yields a CNF variable assignment under which the assumption literals are all simultaneously true; therefore, the corresponding AIG literals can't be simultaneously true either. So to check the equivalence of AIG nodes a and b, we can do two SAT checks after encoding both nodes into CNF: one with assumption $\mathrm{cnf}(a) \wedge \neg\mathrm{cnf}(b)$, and one with assumption $\neg\mathrm{cnf}(a) \wedge \mathrm{cnf}(b)$. If both these checks return UNSAT, we can then conclude that there is no evaluation of the AIG in which the values of nodes a and b differ.

To perform SAT sweeping, we begin with a set of potential equivalences between the nodes, derived by random simulation. We then sweep through the graph in topological order, meaning all of a node's fanins must be processed before we process that node. As we sweep, we build a copy of the graph with redundancies removed, and maintain a mapping from the processed nodes of the input graph to their combinationally-equivalent analogues in the output graph. To sweep a node Q, we first create a new node Q' in the output graph whose fanins are the analogues of the fanins of Q. If Q has no potential equivalences or all potential equivalences occur later in the topological order (and therefore haven't yet been processed), we set Q' as the analogue of Q and continue with the next node. Otherwise let P be the potentially equivalent node earliest in the topological ordering and let P' be its analogue in the output graph. We check using SAT whether Q' and P' are equivalent. We set the analogue of Q to P' if they are equivalent and Q' if not. If the SAT check produces a counterexample (rather than failing due to a solver limit), we simulate the circuit using that counterexample and refine the candidate equivalence classes to account for any newly differentiated pairs of nodes.

We have proved in ACL2 that this algorithm produces a new AIG that is combinationally equivalent to the input AIG. The correctness proof is based on the invariant that the mapping of nodes of the input graph to their analogues in the output graph preserves combinational equivalence. At a given step, we set the mapping for Q either to Q', which is equivalent to Q because it is the same operator applied to fanins which are equivalent by inductive assumption to the fanins of Q, or to P' in the case where P' has been shown by SAT to be equivalent to Q'.

9 Conclusion

This integration of incremental SAT solvers via the IPASIR API is in everyday use for hardware verification at Centaur, largely through the fraiging transform described above. The library and the fraiging algorithm are both available in the ACL2 community books, in directories `centaur/ipasir` and `centaur/aignet`, respectively.

The soundness of this integration is a work in progress. One soundness bug is known to exist, though it doesn't pose a practical risk of undetected false results. We hope to address this problem in future work, though that might require changes to ACL2 itself. Other soundness problems may be revealed with further study, but we hope that the basic approach has the potential to be sound.

References

[1] ACL2 Community (Accessed: 2018): *ACL2+Books Documentation*. Available at `http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/index.html`.

[2] Tomáš Balyo, Armin Biere, Markus Iser & Carsten Sinz (2016): *SAT Race 2015*. *Artificial Intelligence* 241, pp. 45–65, doi:10.1016/j.artint.2016.08.007.

[3] Tomáš Balyo, Marijn J. H. Heule & Matti Järvisalo, editors (2017): *Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions*. *Publication series B* B-2017-1, University of Helsinki, Department of Computer Science. Available at `http://hdl.handle.net/10138/224324`.

[4] Robert S. Boyer, David M. Goldschlag, Matt Kaufmann & J Strother Moore (1991): *Functional instantiation in first-order logic*. In Vladimir Lifschitz, editor: *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, Academic Press, Inc., pp. 7–26, doi:10.1016/B978-0-12-450010-5.50007-4.

[5] Jared Davis & Sol Swords (2013): *Verified AIG Algorithms in ACL2*. In Ruben Gamboa & Jared Davis, editors: Proceedings International Workshop on the *ACL2 Theorem Prover and its Applications*, Laramie, Wyoming, USA , May 30-31, 2013, *Electronic Proceedings in Theoretical Computer Science* 114, Open Publishing Association, pp. 95–110, doi:10.4204/EPTCS.114.8.

[6] Shilpi Goel, Warren A Hunt, Jr. & Matt Kaufmann (2013): *Abstract Stobjs and Their Application to ISA Modeling*. In Ruben Gamboa & Jared Davis, editors: Proceedings International Workshop on the *ACL2 Theorem Prover and its Applications*, Laramie, Wyoming, USA , May 30-31, 2013, *Electronic Proceedings in Theoretical Computer Science* 114, Open Publishing Association, pp. 54–69, doi:10.4204/EPTCS.114.5.

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

[8] Alan Mishchenko, Satrajit Chatterjee, Robert Brayton & Niklas Een (2006): *Improvements to Combinational Equivalence Checking*. In: *Proceedings of the 2006 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '06, ACM, New York, NY, USA, pp. 836–843, doi:10.1145/1233501.1233679.

[9] Yan Peng & Mark R. Greenstreet (2015): *Extending ACL2 with SMT Solvers*. In: *Proceedings Thirteenth International Workshop on the ACL2 Theorem Prover and Its Applications, Austin, Texas, USA, 1-2 October 2015.*, pp. 61–77, doi:10.4204/EPTCS.192.6.

[10] Erik Reeber & Warren A. Hunt (2006): *A SAT-Based Decision Procedure for the Subclass of Unrollable List Formulas in ACL2 (SULFA)*. In Ulrich Furbach & Natarajan Shankar, editors: *Automated Reasoning*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 453–467, doi:10.1007/11814771_38.

[11] J. Sawada & E. Reeber (2006): *ACL2SIX: A Hint used to Integrate a Theorem Prover and an Automated Verification Tool*. In: *2006 Formal Methods in Computer Aided Design*, pp. 161–170, doi:10.1109/FMCAD.2006.3.

[12] Sol Swords & 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.

[13] G. S. Tseitin (1983): *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*, chapter On the Complexity of Derivation in Propositional Calculus, pp. 466–483. Springer, Berlin, Heidelberg, doi:10.1007/978-3-642-81955-1_28.

Using ACL2 in the Design of Efficient, Verifiable Data Structures for High-Assurance Systems[*]

David Hardin[†]
Advanced Technology Center
Rockwell Collins
Cedar Rapids, IA USA
david.hardin@rockwellcollins.com

Konrad Slind
Advanced Technology Center
Rockwell Collins
Bloomington, MN USA
konrad.slind@rockwellcollins.com

Verification of algorithms and data structures utilized in modern autonomous and semi-autonomous vehicles for land, sea, air, and space presents a significant challenge. Autonomy algorithms, *e.g.*, route planning, pattern matching, and inference, are based on complex data structures such as directed graphs and algebraic data types. Proof techniques for these data structures exist, but are oriented to unbounded, functional realizations, which are not typically efficient in either space or time. Autonomous systems designers, on the other hand, generally limit the space and time allocations for any given function, and require that algorithms deliver results within a finite time, or suffer a watchdog timeout. Furthermore, high-assurance design rules frown on dynamic memory allocation, preferring simple array-based data structure implementations.

In order to provide efficient implementations of high-level data structures used in autonomous systems with the high assurance needed for accreditation, we have developed a verifying compilation technique that supports the "natural" functional proof style, but yet applies to more efficient data structure implementations. Our toolchain features code generation to mainstream programming languages, as well as GPU-based and hardware-based realizations. We base the Intermediate Verification Language for our toolchain upon higher-order logic; however, we have used ACL2 to develop our efficient yet verifiable data structure design. ACL2 is particularly well-suited for this work, with its sophisticated libraries for reasoning about aggregate data structures of arbitrary size, efficient execution of formal specifications, as well as its support for "single-threaded objects" — functional datatypes with imperative "under the hood" implementations.

In this paper, we detail our high-assurance data structure design approach, including examples in ACL2 of common algebraic data types implemented using this design approach, proofs of correctness for those data types carried out in ACL2, as well as sample ACL2 implementations of relevant algorithms utilizing these efficient, high-assurance data structures.

1 Introduction

As autonomous systems have matured from laboratory curiosities to sophisticated platforms poised to share our roadways, sea lanes, and airspace, accrediting agencies are faced with the significant challenge of verifying and validating these systems to ensure that they do not constitute a significant societal risk. Leaving aside the issues with the verification and validation of deep learning, even basic autonomy algorithms, *e.g.*, route planning, pattern matching, and inference, are based on complex data structures, such as directed graphs and algebraic data types. Proof techniques for these data structures exist, but are oriented to unbounded, functional realizations, which are not typically efficient in either space or time.

[*]Approved for Public Release, Distribution Unlimited

[†]The views expressed are those of the authors and do not reflect the official policy or position of the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

Shilpi Goel and Matt Kaufmann (Eds.):
ACL2 Theorem Prover and its Applications (ACL2-2018)
EPTCS 280, 2018, pp. 61–76, doi:10.4204/EPTCS.280.5

Autonomous systems designers, on the other hand, generally limit the space and time allocations for any given function, and require that algorithms deliver results within a finite time, or suffer a watchdog timeout. Furthermore, high-assurance design rules, such as mandated by RTCA DO-178C Level A [24] for flight-critical systems, frown on dynamic memory allocation, preferring simple array-based data structure implementations.

In order to provide efficient implementations of high-level data structures used in autonomous and other critical systems with the high assurance needed for accreditation, we have been developing a verification technique that supports the "natural" functional proof style, but yet applies to more efficient data structure implementations. We have used ACL2 to prototype and refine an in-place data structure representation amenable for formal proof, as detailed in Section 6. Applications include path planning on high-level graphs, inference engines for autonomous mission executives, *etc*.

We have significant experience in the formal verification of practical engineering artifacts [7]. Our particular experience with the formal verification of array-based implementation of algebraic data types indicates that these proofs are extremely difficult, whether one takes a theorem proving [9], model checking [10], or symbolic execution [2] approach. The former approach, while yielding a proof for arbitrary array size, easily gets bogged down in supplementary details (even when using list/array "bridging" constructs such as ACL2 single-threaded objects [3]), whereas the latter two approaches suffer from lack of scalability, leading to timeouts except for very small arrays — but unsurprisingly, most practical algorithms operate routinely on thousands of data elements. Therefore, we need a different approach.

We have applied verifying compiler technology to this problem. Algorithms are expressed in a Domain-Aware System Language. A Domain-Aware language is one whose features are informed by the requirements of the domain, such as primitive operations, environmental constraints, assurance and accreditation requirements, and so on, but which is not overtly tied to a particular domain. Our primary goal is to craft a verification-enhanced programming language to support the high assurance requirements of the domains in which it will be used.

2 Verification-Enhanced Programming Languages

A *verification-enhanced programming language* is one in which properties of programs in the language can be formally stated and reasoned about in an integrated environment. Examples of such languages are SPARK/Ada [19], Dafny [16], Guardol [12], and certain C dialects[1] [18,27]. Reasoning environments for C have become highly developed because of the pervasive usage of C in important system infrastructure such as operating systems, cryptography libraries, *etc*.; but C does not support our goal of producing high-level functional correctness proofs.

Once one has constructed a verification-enhanced language, it is relatively simple to create another: after a point, much of the middle and backend processing changes little, leaving only the frontend to be adjusted to the new concrete syntax and type system. This explains the rise of the *intermediate verification language*. For example, Boogie [17] underlies a number of languages, such as VCC and Dafny, and WhyML [5] underlies verification-enhanced versions of C and Ada. Figure 1 illustrates the basic toolchain pattern: an IDE is used to create and edit programs and their specifications; typechecking and semantic checking of programs usually takes place here as well, in order to give good feedback on program construction. The program is then mapped to an abstract syntax tree (AST) representation suitable for code generation/compilation or for mapping into a verification-friendly format. Source-to-

[1]Note that any logic capable of expressing computable functions, *e.g.*, ACL2 [14] or higher order logic, can be regarded as a verification-enhanced programming language; our emphasis here is on more conventional programming languages.

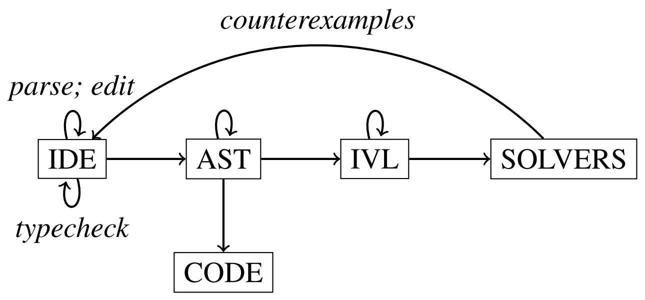


Figure 1: Verification tool design pattern.

source transformations are possible in the AST representation and also in the IVL representation.[2] From the IVL representation, SMT solvers, or other automatic methods can be invoked. Interactive proofs can also be initiated for properties lying outside the domain of automation. In case the backend solvers provide counterexamples, they may be translated into user-friendly format and passed back to the IDE.

We note the following benefits of providing the IVL with a formal operational semantics:

1. The semantics provide a formal basis for individual programs to be proved correct. By use of *decompilation into logic* [20], imperative programs in the operational semantics can be converted to equivalent logic functions by deduction.
2. The formal semantics are the basis for proving the correctness of AST-to-AST transformations.
3. An exciting aspect of the CakeML verified compiler [15] is the existence of a verifying translation from entities in the HOL logic to CakeML programs [21]. As a consequence, HOL functions can be defined and have properties proved about them, then can be automatically translated to CakeML. The correctness of the CakeML compiler ensures that the behavior of the compiled binary version of the function is that of the original logic function. We can further leverage this capability to establish a formal connection between the AST for the original program and the final executable.

3 Modelling Imperative Languages

We will use *DASL* as an example of a verification-enhanced language. DASL (Domain-Aware System Language) is a first order hybrid functional/imperative language with constructs familiar from Ada, ML, Swift and other such languages. The constructible types build on a standard collection of base types: booleans, signed and unsigned integers (both bounded and unbounded), characters, and strings. Arrays and records support aggregation and ML-style recursive datatypes provide tree-shaped data. Programs are built from assignment, sequencing, conditional statements, and (possibly recursive) procedures. The main novelty in the statement language is support for ML-style `match` statements over the construction of datatypes. The concrete syntax is conventional and should not cause any surprises. The instantiation of the verification tool architecture of Figure 1 to DASL is shown in Figure 2.

We chose HOL4 as the environment for the "middle-end" of the DASL compilation toolchain due to its support for higher-order logic, the primary DASL developer's HOL4 expertise, as well as the verified path from executable logic function to binary code provided by the HOL4-based CakeML toolchain.

[2]In some systems, the AST representation and the IVL representation coincide.

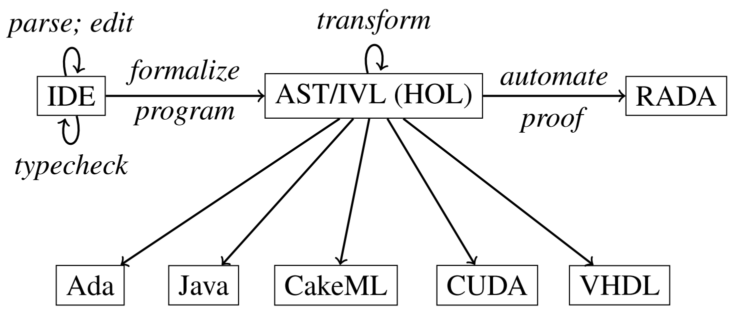


Figure 2: DASL toolchain.

Once in the AST format, source code can be generated for a variety of programming languages. The backend solver is RADA, a SMT-based system for reasoning about recursive programs over algebraic datatypes [23].

3.1 Formal Operational Semantics

The operational semantics of DASL describes program evaluation by a 'big-step' inductively defined judgement saying how statements alter the program state. The formula $\mathsf{STEPS}\,\Gamma\,\mathit{code}\;s_1\;s_2$ says "evaluation of statement *code* in environment Γ beginning in state s_1 terminates and results in state s_2". We have also formalized a small-step semantics and proved equivalence of the two semantics. Note that Γ is an environment binding procedure names to procedure bodies. We follow an approach taken by Norbert Schirmer [25], wherein he constructed a *generic* semantics capturing a large class of sequential imperative programs, and then instantiated the generic semantics to a given programming language.

3.2 Decompilation into Logic

The pioneering work of Myreen [20] introduced the idea of decompiling assembly programs to higher order logic functions; we have adapted his approach to our high-level imperative language. For us, a decompilation theorem has the stylized form

$$\begin{array}{l}\vdash \forall s_1\; s_2.\; \forall x_1 \ldots x_k. \\ \quad s_1.\mathit{proc}.v_1 = x_1 \wedge \cdots \wedge s_1.\mathit{proc}.v_k = x_k \wedge \\ \quad \mathsf{STEPS}\;\Gamma\;\boxed{\mathit{code}}\;(\mathsf{Normal}\;s_1)\;(\mathsf{Normal}\;s_2) \\ \quad \Rightarrow \\ \quad \texttt{let}\;(o_1,\ldots,o_n) = \boxed{f(x_1,\ldots,x_k)} \\ \quad \texttt{in}\; s_2 = s_1\;\texttt{with}\{\mathit{proc}.w_1 := o_1,\ldots,\mathit{proc}.w_n := o_n\}\end{array}$$

which essentially states that evaluation of *code* implements HOL function f. The antecedent $s_1.\mathit{proc}.v_1 = x_1 \wedge \cdots \wedge s_1.\mathit{proc}.v_k = x_k$ binds logic variables $x_1 \ldots x_k$ to the values of program variables $v_1 \ldots v_k$ in state s_1. These values form the input for the so-called *footprint* function f, which delivers the output values $o_1,\ldots,o_n$ that are used to update s_1 to s_2.[3] One can see that a decompilation theorem is a particular kind of Hoare triple. (An explicit Hoare triple approach is used by Myreen in his work.)

[3]In our modelling, a program state is represented by a record containing all variables in the program. The notation $s.\mathit{proc}.v$ denotes the value of program variable v in procedure *proc* in state s. The `with`-notation represents record update.

NB. The footprint function f is automatically synthesized from *code* and the decompilation theorem is proved automatically. In other words, *decompilation is an algorithm*: it always succeeds, provided that all footprint functions arising from the source program terminate.

Decompilation can also be applied to the problem of creating goals from program specifications. A DASL specification sets up a computational context—a state—and asserts that a property holds in that state. In its simplest form, a specification looks like

```
spec name {
  var <local variable declarations>
  in code;
     check property; }
```

where *property* is a boolean expression. A specification declaration is processed as follows. First, suppose that execution of *code* starts normally in s_1 and ends normally in s_2, *i.e.*, assume

$$\mathsf{STEPS}\ \Gamma\ \mathit{code}\ (\mathsf{Normal}\ s_1)\ (\mathsf{Normal}\ s_2).$$

We want to show that *property* holds in state s_2. We decompile *code* to footprint function f and decompile *property* to footprint function g; then, formally, we need to show

$$\begin{array}{l}(\mathtt{let}\ (o_1,...,o_n) = f(x_1,\ldots,x_k) \\ \mathtt{in}\ s_2 = s_1\ \mathtt{with}\ \{name.w_1 := o_1,\ldots,name.w_n := o_n\}) \\ \Rightarrow g\ s_2\end{array}$$

The proof proceeds using facts about f, principally its recursion equations and induction theorem, to show the translated property g holds on values projected from the final state s_2. The original code and property have been freed—by sound deductive steps—from the program state and operational semantics.

4 Data Structure Compilation

Compilation takes a DASL package expressed as a collection of type and procedure declarations, and maps it, when a fixed size declaration is present, to another package where algebraic datatypes have been replaced by array-based representations, and functions over the datatypes have been similarly lowered.

The compilation of data structures is phrased as a source-to-source translation on DASL ASTs. It bears some resemblance to the compiler described in [26]. The target representation was originally used in graph algorithms on GPUs, but also serves quite well for imperative algebraic datatypes.

4.1 Target Representation

Our array-based data structure representation is adapted from work by Harish and Narayanan [13] on efficient graph algorithms for GPUs using the CUDA language; this layout was previously ported to ACL2 single-threaded objects [9]. This design supports graphs with data at vertices and also on edges. Graphs are bounded: both the number of vertices and the number of outgoing edges per vertex are bounded.

One modification we have made to the layout described in the Harish and Narayanan paper [13] is that zero designates a null vertex index or null edge index. We explicitly allocate a zeroth element for each array, and prove that the zeroth elements are unchanged by any array mutator.

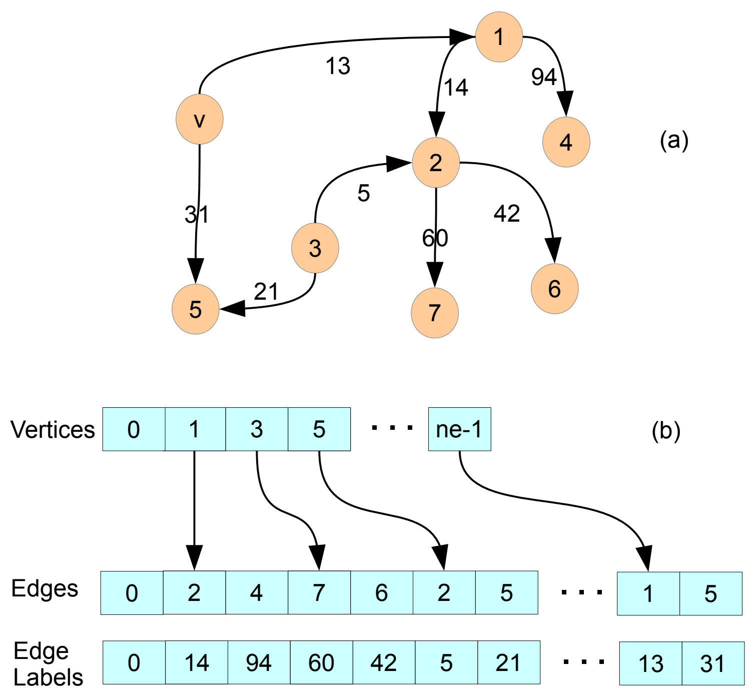


Figure 3: (a) Graph fragment with two edges per vertex; (b) Array-based layout for the graph fragment.

A graph with maximum number of vertices N and maximum number of outgoing edges per vertex M is represented by a seven-tuple $store = (V, D, E, W, Vhd, Vtl, Vcount)$ where

- V is the *vertex* array of length N+1;
- D is the *data* associated with each vertex, requiring an array of length N+1;
- E is the *edge* array of length MN+1;
- W holds the "weight" or "label" data associated with each edge, also an array of length MN+1;
- Vhd is the index of the "head" vertex in V;
- Vtl is the index of the "tail" vertex in V; and
- $Vcount$ is the number of non-zero elements in V.

The vertex array contains indices into the edge array, whereas the edge array contains vertex indices, as shown in Fig. 3 (the data associated with each vertex are not shown, in the interest of clarity). The weight array contains the weight of each edge, and thus is the same size as the edge array. Note that this basic graph tuple can be expanded to include additional data, *e.g.*, keys for keyed data structures.

4.2 Verification

To justify the translation into the array-based representation is an exercise in compiler verification. Our approach uses decompilation into logic. We also want to join the compiler verification results with properties of individual programs. For example, programs over recursive types in embedded systems are typically tail recursive, in order to avoid memory allocation. However, tail recursive programs (and their

$$\begin{tikzcd}
& \boxed{P_{imp}} \arrow[rr, "\mathit{compilation}"] \arrow[d, "\mathit{decompilation}"'] & & \boxed{P_{array}} \arrow[d, "\mathit{decompilation}"'] \\
\boxed{Fn_{logic}} \arrow[r, dash, "\equiv"'] & \boxed{Fn_{imp}} \arrow[rr, dash, "\equiv"'] \arrow[dr, dashed] & & \boxed{Fn_{array}} \arrow[dl, dashed] \\
& & \boxed{\text{CakeML}} &
\end{tikzcd}$$

Figure 4: Verifying translation.

notorious `while`-loop brethren) are harder to reason about than recursive versions. So it is desirable to also have a high-level logical characterization of the program to reason about, provided properties proved at that level can be transported to the program representations.

Figure 4 illustrates the relationships: imperative program P_{imp} over tree-structured data is compiled to program P_{array} over our array-based representation. These are decompiled to Fn_{imp} and Fn_{array}, respectively. In our design, the formal equivalence between Fn_{imp} and Fn_{array} is intended to be automatically proved. The tactics for this are in development. On the other hand, the relationship between high-level logical function Fn_{logic} and Fn_{imp} may require ingenuity to show. The end result is that properties proved of Fn_{logic} can be transported—by use of the equivalence theorems—so that they apply, modulo adjustments to the underlying representation, to the evaluation of P_{array} on the array representation.

Another interesting point is that the translation from P_{imp} to P_{array} is *informal*, *i.e.*, compilation is not achieved by proof steps, and yet decompilation into logic still allows a formal relationship between P_{imp} and P_{array} to be shown.

5 Graphs

The support for imperative functional programming via compilation into the array-based representation offers a useful level of abstraction for many programs. However, there are important applications where the full expressiveness of graphs is needed, for example route planning. In such cases, graphs could certainly be encoded via datatypes, but it is much more appealing to provide a direct representation. Therefore we have added graph constructs as primitives in DASL.

In particular, the declaration

$$\mathsf{graphtype}\ \mathit{name}\ (\mathsf{nodeLabel}, \mathsf{edgeLabel})$$

creates a new type *name* with the specified types of node labels and edge labels. With that declaration, an API for creating, traversing, and updating graphs of type *name* becomes available to the programmer. To exactly specify bounds on the numbers of nodes and edges, the `sized` declaration is extended, thus

$$\mathsf{sized}\ \mathtt{theGraph} : \mathit{name}\ (m, n)$$

declares a global variable `theGraph` holding an empty graph of type *name* with at most *m* nodes, where each node can have at most *n* outgoing edges.

As an example of the use of DASL to implement the sorts of graph algorithms commonly encountered in autonomy and other high-assurance applications, consider the code for depth-first search depicted in Figure 5. The type of graphs handled by this code has unsigned integers at nodes (vertices) and unsigned integers labelling edges; other vertex and edge label types could just as easily been declared. Constants used to bound the graph are also declared. Finally, a fine point of the design is whether to expose the type of vertices. Since vertices are used to index into various data structures, we do show the representation.

```
type vertexLabelTy = uint;
type edgeLabelTy = uint;

graphtype graph (nodeLabel = vertexLabelTy,
                 edgeLabel = edgeLabelTy);
const MAX_NODES : uint = 50000;
const MAX_EDGES : uint = 4;
type vertex = uint;

sized theGraph : graph (MAX_NODES, MAX_EDGES);
```

In this variant of depth-first search, we capture the spanning tree of the depth-first search as we proceed from a starting vertex. We use a binary search tree (BST) of key/value pairs to capture the spanning tree as the depth-first search proceeds, where the keys are the reached vertices, and the values are the immediately preceding vertices for the reached vertices. The BST for the spanning tree serves a dual purpose, as it is also used to check (via the *exists* function) whether a given vertex has been previously encountered during the current search, so that we can cut off search to already-explored sub-graphs.

The search function `DFS_span` is a recursive function whose parameters include a target vertex to be searched for (*target*); the graph *G*; the emerging *spanning_tree* of type BST; and a current *fringe* of unexplored edges, represented as a list of *(vertex, predecessor-vertex)* pairs (this list is conceptually a stack, and could be implemented using a defined DASL stack datatype). We also implement a "driver" function that initializes data structures, defines the initial vertex for the search, *etc*.; this driver is not shown due to space limitations.

The depth-first search in `DFS_span` proceeds as follows. If there are no fringe pairs to consider, we are done, and the (empty) fringe and spanning tree are returned to the caller. If the *target* vertex is in the *spanning_tree*, the search is successful, and we return the fringe and spanning tree to the caller. If there are fringe vertices to explore, we remove a *(v, vpred)* pair from the front of the *fringe* list (using the *rst* rest-of-list function), then check to see if the vertex *v* has already been encountered. If so, we are done with this fringe pair, and so we call `DFS_span` recursively to access the next fringe pair in the list (assuming there are any other pairs left). If *v* has not been encountered before, we add the *(v, vpred)* key/value pair to the *spanning_tree*, add children nodes of *v* to the *fringe* list, utilizing the *explore* function (not shown), then call `DFS_span` to continue the search.

This algorithm statement is compact, and (in our-not-so-objective opinion) quite elegant. The dual use of the binary search tree for both marking visited vertices and recording the spanning tree as the algorithm proceeds is especially satisfying. Note also that a breadth-first search implementation can be readily obtained from the above by changing the *explore* function slightly, adding new fringe vertex pairs to the tail of the *fringe* list, rather than to the head of the list. Note also that a complete spanning tree from a given vertex can be obtained by providing a "bogus" target vertex that doesn't actually exist in the graph (we generally utilize the "null" vertex value of 0 for this purpose). Finally, the algorithm scales to millions of vertices, with tens of edges per vertex, executes quite quickly, and is readily compiled to hardware for even more speed.

```
function DFS_span (vtarget: in vertex, G: in graph,
                   spanning_tree: inout BST,
                   fringe: inout vertex_pair_list) {
  var v, vpred: vertex;
  in
    match fringe {
      -- Out of vertex pairs to process
      'Empty => skip;
      'Node n => {
        -- Found target vertex
        if exists(vtarget, spanning_tree) then
          skip;
        else {
          (v, vpred) := n.elt;
          rest(fringe);
          -- if v already found, on to the next fringe element
          if exists(v, spanning_tree) then
            DFS_span(vtarget, G, spanning_tree, vertex_pair_list);
          else {
            mark(v, vpred, spanning_tree);
            explore(MAX_EDGES, v, G, spanning_tree, vertex_pair_list);
            DFS_span(vtarget, G, spanning_tree, vertex_pair_list);
          }}}}}
```

Figure 5: Depth-First Search in DASL.

6 Use of ACL2 in DASL datatype and graphtype Design

The development of the DASL language, toolchain, and runtime is a complex undertaking, and requires a solid foundation in the form of the basic `datatype` and `graphtype` design and implementation. Early on, we identified the need for a rapid, yet formal, prototyping environment that would allow us to experiment with design alternatives, evaluate these alternatives at scale, and provide initial proofs of correctness for data structure implementations before committing to a final design. Otherwise, we risked proceeding with flawed representations, requiring significant reworking of the DASL toolchain to repair. In short, we needed a "Semantic Laboratory" for DASL development. ACL2 filled the bill admirably:

- ACL2 is the most capable system we know of for the creation, proof, and execution of formal specifications. Using ACL2, we have been able to easily scale our data structure prototype implementations to millions of vertices and edges. We have been able to quickly execute algorithms to exercise these large data structures on concrete test input values, often generated at random, and validate the results of said algorithms operating on our prototype datatypes and graphtypes.

- ACL2 provides *single-threaded objects*, or *stobjs*, that provide functional data structure definitions with destructive "under-the-hood" implementations (subject to basic syntactic restrictions that guarantee that no "old" versions of mutated data structures exist in a given function). Thus, large data structures are not constantly being copied, as they would be for most pure functional implementations, eliminating garbage generation/collection times for stobj data structures. The contents of a stobj data structure also persist in the ACL2 world between events, providing convenient "near-global" variables that can be examined between function invocations, thus greatly aiding the debugging process.

- ACL2 provides sophisticated proof libraries (books) for reasoning about aggregate data structures of arbitrary size, as well as fixed-size integers of various widths.
- Tail recursion in ACL2 combines recursive functional style with efficient compilation to loops.
- ACL2 guards promote a type-like discipline with the added rigor of formal proof. In particular, when we began writing data structure mutators, it became obvious that, in order to call said mutators from other guard-enabled functions, we would first need to prove that the mutators preserved the basic data structure "footprint" predicate provided by the `defstobj` event.
- ACL2's simple packaging facility provides separate namespaces for datatypes/graphtypes.
- All functions admitted to ACL2 must first be proven to terminate. This encourages the ACL2 developer to explicitly consider termination issues when writing functions.
- ACL2 is a mostly-automated theorem prover, and is quite adept at automated inductive proofs.

Thus, we implemented a Rudimentary ACL2 Semantic Laboratory for DASL, which we cheekily refer to as RASL DASL.

Of course, not all of these ACL2 features work together without issue. For example, it is more difficult to perform proofs about tail-recursive functions than their non-tail-recursive counterparts. Guards are not part of the ACL2 logic; if one wants the benefit of guard predicates in a function, one must restate those predicates in the function body. Not all ACL2 books work well in concert. Finally, stobjs are more difficult to reason about than, say, simple Lisp lists. Indeed, when we first starting using tail recursion and stobjs to implement "classic" data structures, we were largely frustrated in our efforts to perform proofs about compositions of functions operating on stobj-based data structures [8]. As often happens with ACL2, however, improvements in ACL2 over time, coupled with ACL2's uncanny ability to instruct the user to produce the kinds of forms that ACL2 "likes", have overcome many of these hurdles. We are now able to routinely obtain correctness proofs for compositions of tail-recursive functions operating on our prototype DASL "in-place" data structures using stobjs, albeit with some user/prover interaction.

6.1 Example datatype: Binary Search Tree

As an example of the ACL2-based datatype prototyping effort, consider the development of a basic Binary Search Tree (BST) datatype. The ACL2 stobj definition for this type is given in Figure 6. (Note that graphtypes are defined similarly, as the underlying data structure representation is explicitly designed to represent graphs.) The declaration of the `Obj` stobj is followed by a number of basic functions (adding a vertex, deleting a vertex, *etc.* — not shown due to space constraints), as well as `defthm` forms that provide basic lemmas about the components of the datatype (*e.g.*, that the key array is an `integer-listp`, or that the result of updating the val array continues to satisfy the `Objp` predicate that is synthesized by the `defstobj` form depicted in Figure 6).

In DASL datatype/graphtype prototyping, we generally place the basic structure definitions, *etc.* in one file, and the higher-level functions and theorems (*e.g.*, functions that insert a new key/value pair, delete a key/value pair, check to see if a key is in the BST, return the value for a given key, and so on) in a second file. That way, similar datatypes that share a common structure can reuse the file of basic definitions, with slight modifications. As an example of the sort of higher-level functions we typically define, consider the `getVal` function of Figure 7, along with its shorthand macro form `getV`. `getVal` follows the classic textbook definition, with a bit of the underlying ACL2 data representation showing through. Note that termination for this function is not assured in general; thus, we add a `count` parameter that is decremented on each recursive call of `getVal`. Fortunately, we have a convenient value at hand to serve as a reasonable initial value for `count`, namely the number of vertices in the BST.

```
(in-package "BST")

(defconst *MAX_VTX* 65535)
(defconst *MAX_VTX1* (1+ *MAX_VTX*)) ;; 2**16
(defconst *MAX_EDGES_PER_VTX* 2)
(defconst *MAX_EDGE* (* *MAX_VTX* *MAX_EDGES_PER_VTX*))
(defconst *MAX_EDGE1* (1+ (* *MAX_VTX* *MAX_EDGES_PER_VTX*)))
(defconst *MAX_EDGE_MINUS* (1+ (- *MAX_EDGE* *MAX_EDGES_PER_VTX*)))

(defstobj Obj
;; padding -- keeps ACL2 from turning (nth *VTXHD* Obj) into (car Obj)
  (pad :type t :initially 0)
  (vtxHd :type (integer 0 65535) :initially 0)
  (vtxTl :type (integer 0 65535) :initially 0)
  (vtxCount :type (integer 0 65535) :initially 0)
;; (V) This contains a pointer to the edge list for each vertex
  (vtxArr :type (array (integer 0 131069) (*MAX_VTX1*)) :initially 0)
;; (K) Keys for each vertex
  (keyArr :type (array (integer 0 *) (*MAX_VTX1*)) :initially 0)
;; (D) Data Value array
  (valArr :type (array (integer 0 *) (*MAX_VTX1*)) :initially 0)
;; (E) This contains pointers to the vertices that each edge is attached to
  (edgeArr :type (array (integer 0 65535) (*MAX_EDGE1*)) :initially 0)
 :inline t)
```

Figure 6: Binary Search Tree stobj declaration.

```
(defun getVal (count key vtx Obj)
  (declare (xargs :stobjs Obj
                  :guard (and (natp count) (natp key) (natp vtx))))
  (cond
   ((not (mbt (Objp Obj))) 0)    ;; Only positive values stored. 0 = 'null'
   ((not (mbt (natp count))) 0)
   ((not (mbt (natp key))) 0)
   ((not (mbt (natp vtx))) 0)
   ((zp count) 0)
   ((zp key) 0)
   ((zp vtx) 0)
   ((> vtx *MAX_VTX*) 0)
   ((mbe :logic (zp (vtxCount Obj))
         :exec (int= (vtxCount Obj) 0)) 0) ;; no vertices
   ((zp (keyArri vtx Obj)) 0)
   ((< key (keyArri vtx Obj))
    (getVal (1- count) key (edgeArri (left (vtxArri vtx Obj)) Obj) Obj))
   ((> key (keyArri vtx Obj))
    (getVal (1- count) key (edgeArri (right (vtxArri vtx Obj)) Obj) Obj))
   ;; (= key (keyArri vtx Obj))
   (t (valArri vtx Obj))))

(defmacro getV (key Obj)
  `(getVal (vtxCount ,Obj) ,key (vtxHd ,Obj) ,Obj))
```

Figure 7: Example DASL-derived Binary Search Tree function prototyped in ACL2.

```
(defun dfs_span (count vtarget gObj BST::Obj DLPR::Obj)
  (declare (xargs :stobjs (gObj BST::Obj DLPR::Obj)
                  :guard (and (natp count) (natp vtarget))))
  (cond
   ((not (and (mbt (natp count))
              (mbt (natp vtarget))
              (mbt (gObjp gObj))
              (mbt (BST::Objp BST::Obj))
              (mbt (DLPR::Objp DLPR::Obj)))) (mv BST::Obj DLPR::Obj))
   ((zp count) (mv BST::Obj DLPR::Obj))
   ((> vtarget *MAX_VTX*) (mv BST::Obj DLPR::Obj))
   ;; Found target vertex
   ((BST::existp vtarget BST::Obj) (mv BST::Obj DLPR::Obj))
   ((zp (DLPR::ln DLPR::Obj)) (mv BST::Obj DLPR::Obj))
   ;; Grab (v, vpred) pair from head of list
   (t (mv-let (v vpred) (DLPR::nthelem 0 DLPR::Obj)
        (cond
         ((not (posp v)) (mv BST::Obj DLPR::Obj))
         ((> v *MAX_VTX*) (mv BST::Obj DLPR::Obj))
         ((not (posp vpred)) (mv BST::Obj DLPR::Obj))
         ((> vpred *MAX_VTX*) (mv BST::Obj DLPR::Obj))
         ((BST::existp v BST::Obj)
          (seq2 BST::Obj DLPR::Obj
                (BST::nop BST::Obj)
                (DLPR::rst DLPR::Obj)
                (dfs_span (1- count) vtarget gObj BST::Obj DLPR::Obj)))
         (t (seq2 BST::Obj DLPR::Obj
                  (mark v vpred BST::Obj)
                  (seq DLPR::Obj
                       (DLPR::rst DLPR::Obj)
                       (explore *MAX_EDGES_PER_VTX* v gObj BST::Obj DLPR::Obj))
                  (dfs_span (1- count) vtarget gObj BST::Obj DLPR::Obj))))))))
```

Figure 8: Depth-First Search prototyped in ACL2.

7 Results

As an example of the sorts of algorithms we are able to successfully prototype using the RASL DASL environment, consider the hand-translated ACL2 version of the Depth-First Search algorithm of Figure 5, depicted in Figure 8. The ACL2 version utilizes a graphtype (`gObj`, taking on the role of the DASL `G` parameter) and two datatypes: the Binary Search Tree previously discussed (`BST::Obj`, corresponding to the DASL `spanning_tree` parameter), and a doubly-linked list (`DLST::Obj`, acting as the DASL `fringe` parameter). We employ a doubly-linked list, which provides access/update from either end of the list, for the `fringe` parameter because it was convenient to also use the same datatype for the implementation of the Breadth-FIrst Search algorithm (in fact, `dfs_span` and `bfs_span` are functionally identical; the difference in behavior between BFS and DFS is solely due to whether the `explore` function adds the vertex pairs it finds to the front (DFS) or back (BFS) of the list.

As in the BST `getVal` function, the termination of the `dfs_span` function is not explicitly assured, so we again add a `count` parameter that is decremented with each recursive call. The number of vertices in the graph multiplied by the maximum number of edges per vertex serves as a reasonable initial value for `count`. Another unique feature of the ACL2 code requiring some explanation is the `seq2` macro.

We often use J Moore's `seq` macro when working with stobjs, as it eliminates much of the `let` binding "clutter", allowing one to simply write one stobj-manipulating expression after another within the scope of the `seq`. With two stobjs to update and return, the `mv-let` binding and `mv` return "clutter" can get much worse. Thus, we developed a quick-and-dirty macro, `seq2` to do the analogous decluttering job for two stobjs. `seq2` takes as parameters the two stobjs that are to be updated, the updating s-expression for the first stobj, the updating s-expression for the second stobj, and the s-expression that one wishes to return using `mv`. This works well enough in practice, although sometimes only one of the two stobjs one wishes to return using `mv` are actually modified within the scope of the `seq2`. In this case, one must provide some "no-op" s-expression for the unmodified stobj.

Other than these few changes, and a slight rearrangement to concentrate the stobj updates into one contiguous section of the code, the ACL2 version tracks the DASL code fairly well, and the elegance of the algorithm is still apparent.

To date, we have written a number of autonomy-relevant algorithms in DASL after prototyping in RASL DASL, including tree search, graph search, Dijkstra's all-pairs shortest path algorithm, and unify/substitute; as well as a number of supporting data structures, such as priority queues, stacks, singly- and doubly-linked lists, queues, deques, *etc*. We have used our verification environment to state and prove properties of many of these algorithms, starting with basic well-formedness, and proceeding to full functional correctness. Finally, we have generated code for many of these examples using the DASL compiler, and have validated the code generation via testing. As expected, the array-based form leads to efficient execution, scales well to millions of vertices with tens of edges per vertex, and enables hardware-based and GPU-based execution.

8 Related Work and Future Work

We first acknowledge the influence that ACL2 single-threaded objects [3] have had on the overall DASL philosophical approach of providing functional specifications with efficient, "under-the-hood" implementations. We have learned much from the ACL2 developers, particularly the importance of efficient, executable formal specifications.

The Boogie [17] and WhyML/Why3 [5] systems are paradigmatic IVLs, supporting highly developed verification-enhanced languages. The main point of departure with our work is that Boogie and WhyML are programming languages, while the IVL for DASL is higher order logic. Verification-enhanced versions of C are supported by Boogie and Why3, and also by Appel's Hoare Logic for CompCert C [1], which is derived from the operational semantics of CompCert in Coq. The AutoCorres tool [6] arose out of the seL4.verified effort; it translates ASTs from a parser for the C dialect used in seL4 to Schirmer's SIMPL theory in Isabelle/HOL and helps automate much of the Separation Logic used.

Chlipala's Bedrock system [4] also utilizes higher order logic as an IVL; in particular, he uses Coq as the substrate on which to build intermediate and lower-level languages and prove the correctness of transformations. Finally, the work of O'Leary and Russinoff on the formalization of C subsets for hardware design in ACL2 [22] encouraged us to consider a formalized System language.

DASL is a direct descendent of the Guardol Domain-Specific Language for Cross-Domain Systems [12]. Our experience with Guardol convinced us of the efficacy of taking a Domain-Aware approach to language development for various high-assurance domains. Not surprisingly, DASL and Guardol share a number of syntactic and semantic features. DASL also shares a great deal of the verification-oriented tool infrastructure pioneered on the Guardol effort, such as the logic-based IVL, the RADA backend tree solver, as well a VHDL code generator. The latter capability allowed us to synthesize a formally proven high-level regular expression pattern matcher in inexpensive FPGA hardware that performed at Gigabit Ethernet line speeds [11]; these results prompted us to continue to develop VHDL code generation for DASL. DASL distinguishes itself from Guardol mainly in the `datatype` and `graphtype` declarations, the `sized` declaration, and the attendant syntactic restrictions that allow for compilation to an efficient in-place data structure representation.

In future work, we will continue to develop the DASL toolchain, focusing on code generation for CakeML, but also improving code generation for Ada, Java, VHDL, and other target languages. We will also continue to refine the language syntax, focusing on ways to conveniently add extensions such as grammar specifications and rulebases. Regular expressions are introduced currently by way of a simple `regex_match` intrinsic function, which accepts a regular expression as a string parameter, but this method is unlikely to scale well to grammar rules, *etc*. Finally, we will continue to expand on the suite of applications implemented in DASL, especially in the areas of autonomy algorithms, as well as data filtering/transformation for cyber-resilient systems.

9 Conclusion

We have developed a verifying compilation technique for a domain-aware programming language for autonomy and other high-assurance applications that combines efficient implementations of high-level data structures used in autonomous systems with the high assurance needed for accreditation. Our system supports a “natural” functional proof style, yet applies to more efficient data structure implementations. Our toolchain features code generation to mainstream programming languages, as well as GPU-based and hardware-based realizations. We base the Intermediate Verification Language for our toolchain upon higher-order logic. By giving program execution a formal semantics, claims about program behavior can be mathematically proven, and source-to-source transformations of the intermediate form can be proven as well. Thus, proofs about high-level programs over high-level data structures can be carried out while automatically ensuring a formal, proved connection to the low-level efficient implementation. We have also demonstrated that when the IVL is higher order logic, verified code generation is possible via the facilities of the CakeML verified compiler.

We utilized ACL2 to develop our efficient yet verifiable data structure design. ACL2 is particularly well-suited for this task, with its sophisticated libraries for reasoning about aggregate data structures of arbitrary size, efficient execution of formal specifications, and its support for single-threaded objects, not to mention its strength as an automated inductive prover. We described our high-assurance data structure design approach in ACL2, presented ACL2 examples of common algebraic data types implemented using this design approach, discussed proofs of correctness for those data types carried out in ACL2, as well as sample ACL2 implementations of relevant algorithms that utilize these efficient, high-assurance data structures. In summary, this ACL2-based development activity has produced a performant, as well as highly-assured, data structure design for critical applications, and we continue to use this ACL2-based design environment to prototype new applications for modern high-assurance systems, such as lexers and parsers for data interchange formats such as JSON, inference engines, and the like.

10 Acknowledgments

We thank the anonymous referees for their helpful comments. This work was sponsored in part by the Defense Advanced Research Projects Agency (DARPA).

References

[1] Andrew W. Appel (2014): *Program Logics for Certified Compilers*. Cambridge University Press, doi:10.1017/CBO9781107256552.

[2] Jason Belt, John Hatcliff, Robby, Patrice Chalin, David S. Hardin & Xianghua Deng (2011): *Bakar Kiasan: Flexible Contract Checking for Critical Systems using Symbolic Execution*. In: *Proceedings of the Third NASA Formal Methods Symposium (NFM 2011)*, pp. 58 – 72, doi:10.1007/978-3-642-20398-5_6.

[3] Robert S. Boyer & J Strother Moore (2002): *Single-Threaded Objects in ACL2*. In: *Practical Aspects of Declarative Languages, 4th International Symposium, PADL 2002*, *LNCS* 2257, Springer, pp. 9–27, doi:10.1007/3-540-45587-6_3.

[4] Adam Chlipala (2013): *The Bedrock Structured Programming System: Combining Generative Metaprogramming and Hoare Logic in an Extensible Program Verifier*. In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, ACM Press, pp. 391–402, doi:10.1145/2500365.2500592.

[5] Jean-Christophe Filliâtre & Andrei Paskevich (2013): *Why3 — Where Programs Meet Provers*. In Matthias Felleisen & Philippa Gardner, editors: *Proceedings of the 22nd European Symposium on Programming*, *LNCS* 7792, Springer, pp. 125–128, doi:10.1007/978-3-642-37036-6_8.

[6] David Greenaway, Japheth Lim, June Andronick & Gerwin Klein (2014): *Don't Sweat the Small Stuff: Formal Verification of C Code Without the Pain*. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, ACM Press, pp. 429–439, doi:10.1145/2594291.2594296.

[7] David S. Hardin, editor (2010): *Design and Verification of Microprocessor Systems for High-Assurance Applications*. Springer, doi:10.1007/978-1-4419-1539-9.

[8] David S. Hardin & Samuel S. Hardin (2009): *Efficient, Formally Verifiable Data Structures using ACL2 Single-Threaded Objects for High-Assurance Systems*. In S. Ray & D. Russinoff, editors: *Proceedings of the Eighth International Workshop on the ACL2 Theorem Prover and its Applications*, ACM Press, pp. 100 – 105, doi:10.1145/1637837.1637853.

[9] David S. Hardin & Samuel S. Hardin (2013): *ACL2 Meets the GPU: Formalizing a CUDA-based Parallelizable All-Pairs Shortest Path Algorithm in ACL2*. In R. Gamboa & J. Davis, editors: *Proceedings of the 11th International Workshop on the ACL2 Theorem Prover and its Applications*, 114, EPTCS, pp. 127 – 142, doi:10.4204/EPTCS.114.10.

[10] David S. Hardin, T. Douglas Hiratzka, D. Randolph Johnson, Lucas G. Wagner & Michael W. Whalen (2009): *Development of Security Software: A High Assurance Methodology*. In K. Breitman & A. Cavalcanti, editors: *Proceedings of the 11th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering (ICFEM'09)*, Springer, pp. 266 – 285, doi:10.1007/978-3-642-10373-5_14.

[11] David S. Hardin, Konrad L. Slind, Mark A. Bortz, James Potts & Scott Owens (2016): *A High-Assurance, High-Performance Hardware-Based Cross-Domain System*. In: *Computer Safety, Reliability, and Security - 35th International Conference, SAFECOMP 2016*, *LNCS* 9922, Springer, pp. 102–113, doi:10.1007/978-3-319-45477-1_9.

[12] David S. Hardin, Konrad L. Slind, Michael W. Whalen & Tuan-Hung Pham (2012): *The Guardol Language and Verification System*. In: *Proceedings of TACAS*, *LNCS* 7214, Springer, pp. 18–32, doi:10.1007/978-3-642-28756-5_3.

[13] Parwan Harish & P.J. Narayanan (2007): *Accelerating Large Graph Algorithms on the GPU using CUDA*. In: *IEEE High Performance Computing – HiPC 2007*, *LNCS* 4873, Springer, pp. 197–208, doi:10.1007/978-3-540-77220-0_21.

[14] Matt Kaufmann, Panagiotis Manolios & J Strother Moore (2000): *Computer-Aided Reasoning: An Approach*. Texts and Monographs in Computer Science, Kluwer Academic Publishers, doi:10.1007/978-1-4615-4449-4.

[15] Ramana Kumar, Magnus O. Myreen, Michael Norrish & Scott Owens (2014): *CakeML: A Verified Implementation of ML*. In: *POPL '14: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM Press, pp. 179–191, doi:10.1145/2535838.2535841.

[16] K. Rustan M. Leino (2013): *Developing Verified Programs with Dafny*. In: *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, IEEE Press, pp. 1488–1490, doi:10.1109/icse.2013.6606754. Available at `http://dl.acm.org/citation.cfm?id=2486788.2487050`.

[17] K. Rustan M. Leino & P. Ruemmer (2010): *A Polymorphic Intermediate Verification Language: Design and Logical Encoding*. In: *Proceedings of TACAS*, *LNCS* 6015, pp. 312–327, doi:10.1007/978-3-642-12002-2_26.

[18] Xavier Leroy (2009): *Formal verification of a realistic compiler*. *Communications of the ACM* 52(7), pp. 107–115, doi:10.1145/1538788.1538814.

[19] John W McCormick & Peter C. Chapin (2015): *Building High Integrity Applications with SPARK*. Cambridge University Press, doi:10.1017/cbo9781139629294.

[20] Magnus Myreen (2009): *Formal verification of machine-code programs*. Ph.D. thesis, University of Cambridge.

[21] Magnus O. Myreen & Scott Owens (2014): *Proof-producing Translation of Higher-order logic into Pure and Stateful ML*. *Journal of Functional Programming* 24(2-3), pp. 284–315, doi:10.1017/S0956796813000282.

[22] John W. O'Leary & David M. Russinoff (2014): *Modeling Algorithms in SystemC and ACL2*. In: *Proceedings of the 12th International Workshop on the ACL2 Theorem Prover and its Applications*, 152, EPTCS, pp. 145–162, doi:10.4204/EPTCS.152.12. Available at `https://arxiv.org/pdf/1406.1565.pdf`.

[23] Tuan-Hung Pham (2014): *Verification of Recursive Data Types using Abstractions*. Ph.D. thesis, Department of Computer Science and Engineering, University of Minnesota.

[24] RTCA Committee SC-205 (2015): *DO-178C Software Considerations in Airborne Systems and Equipment Certification*. Available at `https://my.rtca.org/nc__store?search=DO-178C`.

[25] Norbert Schirmer (2006): *Verification of sequential imperative programs in Isabelle/HOL*. Ph.D. thesis, TU Munich.

[26] Andrew Tolmach & Dino P. Oliva (1998): *From ML to Ada: Strongly-typed Language Interoperability via Source Translation*. *Journal of Functional Programming* 8(4), pp. 367 – 412, doi:10.1017/s0956796898003086.

[27] Harvey Tuch, Gerwin Klein & Michael Norrish (2007): *Types, Bytes, and Separation Logic*. In: *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, ACM Press, pp. 97–108, doi:10.1145/1190216.1190234.

Adding 32-bit Mode to the ACL2 Model of the x86 ISA

Alessandro Coglio
Kestrel Technology LLC, Palo Alto, CA (USA)
coglio@kestreltechnology.com

Shilpi Goel
Centaur Technology, Inc., Austin, TX (USA)
shilpi@centtech.com

The ACL2 model of the x86 Instruction Set Architecture was built for the 64-bit mode of operation of the processor. This paper reports on our work to extend the model with support for 32-bit mode, recounting the salient aspects of this activity and identifying the ones that required the most work.

1 Motivation and Contributions

A formal model of the ISA (Instruction Set Architecture) of the pervasive x86 processor family can be used to clarify and disambiguate its informal documentation [15], as well as support the verification and formal analysis of existing binary programs (including malware), the verified compilation of higher-level languages to machine code, the synthesis of correct-by-construction binary programs from specifications, and the verification that micro-architectures correctly implement the ISA. Furthermore, if the model is efficiently executable, it can be used as a high-assurance simulator, e.g. as a test oracle.

The ACL2 model of the x86 ISA [1, `:doc x86isa`] [11] is one such model. It includes a substantial number of instructions and has been used to verify several non-trivial binary programs. Its fast execution has enabled its validation against actual x86 processors on a large number of tests.

Prior to the work described in this paper, the model supported the simulation and analysis of 64-bit software only (which includes modern operating systems and applications), and was used to verify several 64-bit programs. We refer to this pre-existing model of the x86 ISA as the *64-bit model*.

However, legacy 32-bit software is still relevant. Many applications are still 32-bit, running alongside 64-bit applications in 64-bit operating systems. Most malware is 32-bit [16]; as part of a project to detect malware variants via semantic equivalence checking by symbolic execution, we were given a large corpus of known Windows malware for experimentation, and found indeed that it mainly contained 32-bit executables.[1] Verifying the 32-bit portion of a micro-architecture requires a model of the 32-bit ISA. Also, the aforementioned verified compilation of higher-level languages to machine code and synthesis of correct-by-construction binary programs from specifications, while presumably generally oriented towards 64-bit code, may occasionally need to target 32-bit code, e.g., to interoperate with legacy systems.

Thus, we worked on extending the 64-bit model with 32-bit support: all the non-floating-point instructions in the 64-bit model have been extended from 64 bits to 32 bits; furthermore, a few 32-bit-only instructions have been added. However, the 32-bit extensions have not been validated against actual x86 processors yet as thoroughly as the 64-bit portions of the model. Work on verifying 32-bit programs using this model has also started but is too preliminary to report. We refer to this extended model of the x86 ISA as the *extended model*.

This paper reports on our work to extend the 64-bit model. Section 2 provides some background on the x86 ISA and on the 64-bit model. Section 3 recounts the salient aspects of the 32-bit extensions to the model, including how they were carried out and which ones required the most work. Related and

[1]This statement refers to a project at Kestrel Technology.

future work are discussed in Section 4 and Section 5. The 64-bit model was developed by the second author [11], and the 32-bit extensions were developed by the first author (with some help and advice from the second author); this suggests that the model is extensible by third parties.

2 Background

We present a very brief overview of the x86 ISA that is relevant to this paper in Section 2.1. In Section 2.2, we describe some features of the 64-bit model of the x86 ISA.

2.1 x86 ISA: A Brief Overview

Intel's modern x86 processors offer various modes of operation [15, Volume 1, Section 3.1]. The IA-32 architecture supports 32-bit computing by giving access to 2^{32} bytes of memory, and it provides three modes:

1. *Real-address mode*, which is the x86 processor's mode upon power-up or reset.
2. *Protected mode*, which is informally referred to as the "32-bit mode"; this mode can also emulate the real-address mode if its *virtual-8086 mode* attribute is set.
3. *System management mode*, which is used to run firmware to obtain fine-grained control over system resources to perform operations like power management.

The Intel 64 architecture added the *IA-32e mode*, which has two sub-modes:

1. *64-bit mode*, which gives access to 2^{64} bytes of memory, and thus, allows the execution of 64-bit code.
2. *Compatibility mode*, which allows the execution of legacy 16- and 32-bit code inside a 64-bit operating system; in this sense, it is similar to 32-bit protected mode vis-à-vis application programs.

The 64-bit model of the x86 ISA, discussed briefly in Section 2.2, specified only the 64-bit sub-mode of IA-32e mode. In the subsequent sections, we focus on 32-bit protected mode and compatibility mode. Modeling the rest of the modes is a task for the future (see Section 5).

IA-32e and 32-bit protected modes are the most used operating modes of x86 processors today. A big difference between these modes is in their memory models — we first briefly discuss the memory organization on x86 machines. There are two main types of memory address spaces: *physical address space* and *linear address space*. The physical address space refers to the processor's main memory; it is the range of addresses that a processor can generate on its address bus. The linear address space refers to the processor's linear memory, which is an abstraction of the main memory (see paging below). Programs usually do not access the physical address space directly; instead, they access the linear address space.

Segmentation and *paging* are two main facilities on x86 machines that manage a program's view of the memory; see Figure 1. When using segmentation, programs traffic in *logical addresses*, which consist of a *segment selector* and an *effective address*. The x86 processors provide six (user-level) segment selector registers — `CS`, `SS`, `DS`, `ES`, `FS`, and `GS`. A segment selector points to a data structure, called a *segment descriptor*, that contains, among other information, the *segment base*, which specifies the starting address of the segment, and the *segment limit*, which specifies the size of the segment. As an optimization, the processor caches information from this data structure in the *hidden* part of these registers whenever a selector register is loaded. The segment base is added to the effective address to obtain the linear address. The upshot of this is that the linear address space is divided into a group of segments. An operating system can assign different sets of segments to different programs or even different segments to the same program (e.g., separate ones for code, data, and stack) to enforce protection and non-interference. It can also assign the same set of segments to different processes to enable sharing and communication. The processor makes three main kinds of checks when using segmentation: (1) limit checks, which cause an exception if code attempts to access memory locations outside the segment; (2) type checks, which cause an exception if code uses a segment in some unintended manner (e.g., an attempt is made to write to a read-only data segment); and (3) privilege checks, which cause an exception if code attempts to access a segment which it does not have permission to access.

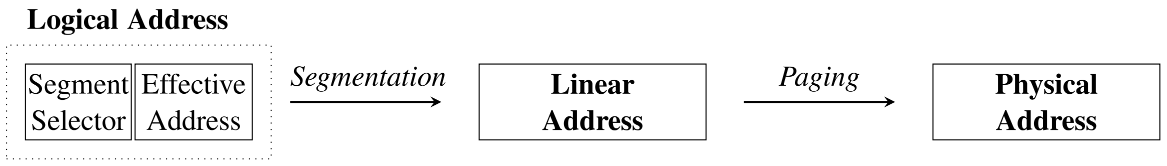


Figure 1: Memory Addresses

Paging is used to map linear addresses to physical addresses. It is conventional to define this mapping in such a manner that a larger linear address space can be simulated by a smaller physical address space. Each segment is divided into smaller pages, which can be resident either in the physical memory (if the page is in use) or on the disk (if the page is not currently in use). An operating system can swap pages back and forth from the physical memory and the disk; thus, paging supports a *virtual* memory environment. There are three kinds of paging offered by x86 processors: 32-bit (translates 32-bit linear addresses to up to 40-bit physical addresses), PAE (translates 32-bit linear addresses to up to 52-bit physical addresses), and 4-level (translates 48-bit linear addresses to up to 52-bit physical addresses).

In 64-bit mode, a logical address, an effective address, and a linear address for a memory location are usually the same because all segment bases, except those for two out of six segment selectors (i.e., `FS` and `GS`), are treated as zero. Moreover, limit checking and a certain kind of type checking are not performed in this mode. Thus, segmentation is effectively a legacy feature in 64-bit mode. However, in 32-bit and compatibility modes, segmentation is used in its full glory.

32-bit protected and IA-32e modes also differ with regard to paging. In 32-bit protected mode, 32-bit and PAE paging are used to translate 32-bit linear addresses. In both sub-modes of IA-32e mode, only 4-level paging is used. However, since 4-level paging translates only 48-bit linear addresses and IA-32e mode has 64-bit linear addresses, this mode has a notion of *canonical addresses*. Canonical addresses are 64-bit linear addresses whose bits 63 through 47 are identical — either all zeroes or all ones. Additionally, since compatibility mode traffics in 32-bit addresses, bits 47 through 32 of a 64-bit linear address are treated as zero. Any attempt to map a non-canonical linear address to a physical address in IA-32e mode will lead to an exception.

Of course, there are many other differences between 32-bit and 64-bit modes. Even instruction decoding differs in these modes. An x86 instruction has a variable length up to 15 bytes: the *opcode* bytes in the instruction determine which operation is to be performed; the other instruction bytes act as size modifiers, or specify the location of the operands, etc. 64-bit instructions can have additional bytes preceding an opcode, called the `REX` prefixes, which can modify the operation's sizes or operands' locations. In 32-bit mode, these bytes actually correspond to the increment/decrement (`INC`/`DEC`) opcodes. Task management is another big example — a task can consist of a program, a process, or an interrupt or exception handler, and it is executed and/or suspended by the processor as an atomic unit. Task management is essentially the software's responsibility in 64-bit mode, but it is handled by hardware in 32-bit protected mode. Generally speaking, 32-bit mode and compatibility sub-mode support many legacy features, and as such, they are less "streamlined" than the newer 64-bit mode.

We refer the reader to Intel's official documentation [15] for more information about the x86 ISA.

2.2 `x86isa`: A Model of the 64-bit Mode

Prior to this work, the `x86isa` library in the ACL2 community books contained the specification of a significant subset of 64-bit sub-mode of IA-32e mode. This 64-bit model follows the classical *interpreter style of operational semantics* [18, 19] — it can be described by the following main components:

- a machine state consisting of registers, memory, flags, etc.;
- instruction semantic functions that define the behavior of x86 instructions in terms of updates made to the machine state;
- a step function that fetches an x86 instruction from the memory, decodes it till the opcode byte(s) are read, and then executes it by calling a top-level opcode dispatch function — this dispatch function is basically a giant case statement that calls the appropriate instruction semantic function corresponding to the opcode; and
- a run function, which is the top-level interpreter that calls the step function iteratively.

The 64-bit model offers two views of x86 machines: the *system-level* view — when the `app-view` field in the machine state is `nil`, and the *application-level* view — when the `app-view` field is not `nil`. The system-level view is intended to specify the x86 ISA as accurately as possible. In this view, one can verify system programs that have access to system state, such as the memory management data structures. The application-level view can be used to simulate and verify application programs in the same sort of environment that programmers use to develop such programs. In this view, the operating assumption is that the underlying operating system and system features of the ISA behave as expected. For instance, unlike the system-level view, this view does not include the model of paging — it offers the linear memory abstraction instead of the physical memory. Also, the application-level view does not contain the specification of system instructions (which are present in the system-level view), e.g., instruction `LGDT`, which manipulates the segmentation data structures. However, the application-level view does specify 64-bit segmentation — i.e., `FS`/`GS`-based logical to linear address translation,[2] — but here it makes the following assumption: the hidden parts of these segment selectors are assumed to contain the correct segment base addresses. This assumption is reasonable because these hidden parts of the registers are automatically populated by the processor when the registers are loaded, and the instructions that load these registers are system-level instructions — thus, it is assumed that an operating system routine correctly loaded these registers. These views aim to mitigate the trade-off between verification utility and verification effort. Of course, if such assumptions that come with the application-level view's higher-level of abstraction are unacceptable to a user, he or she can operate in the system-level view for *all* programs.

This model has been used to do proofs about both application and system programs. Some examples of application programs verified using the model are a population count program, a word-count program, and a data-copy program. All of these programs except population count were verified by using the lemma libraries that are included in the `x86isa` library; the population count program was verified completely automatically using the `GL` library [26, 27]. The 64-bit model is also compatible with the Codewalker library [2]; Codewalker has been used to reason about small application programs like the x86 machine-code routine to compute the factorial of a number. An example of a system program verified using the `x86isa` library is zero-copy, which efficiently copies data from a source to a destination memory location by modifying the memory management data structures (i.e., using the *copy-on-write*

[2]Recall from Section 2.1 that for other segment selectors, the linear and logical addresses are the same in 64-bit mode.

technique).

The x86isa library uses ACL2 features like abstract stobjs [13] to provide both reasoning and execution efficiency. The simulation speed of the 64-bit model is around 300,000 instructions/second in the system-level view and 3 million instructions/second in the application-level view.[3] As a direct consequence, it has been possible to extensively validate the 64-bit model by running co-simulations against Intel's x86 machines, thereby increasing confidence in the model's accuracy. A goal of the x86isa library is to make the x86 ISA specification accessible to users, and as such, these books are extensively documented [1, :doc x86isa] — both from a user and a developer's point of view.

For more about the 64-bit model and its use in 64-bit code analysis, we refer an interested reader elsewhere — see [12] for a more detailed overview and [11, 14] for an in-depth report.

3 Extension of the Model

In this section, when we refer to code running in *32-bit mode*, we mean *application* software running in either 32-bit protected mode or compatibility sub-mode of IA-32e mode.

3.1 Challenges

Extending the model to 32-bit mode was not simply a matter of generalizing the sizes of the operands and addresses manipulated by the instructions. As explained in Section 2.1, 32-bit mode is more complicated than 64-bit mode, due to the legacy features that it provides.

In particular, in 32-bit mode, memory accesses are more complicated than in 64-bit mode: 32-bit mode uses segmentation, which is almost completely disabled in 64-bit mode. The 64-bit model made, throughout, the reasonable assumption that the effective addresses in instruction pointer, stack pointer, addressing mode base registers, etc. were also linear addresses — the only exception was when FS and GS segment selectors were used, in which case their bases were explicitly added to effective addresses to get the corresponding linear addresses. Effective/linear addresses were checked to be canonical as needed. This had to be generalized in the extended model: effective and linear addresses had to be differentiated, and segment bound checks had to be performed, before adding segment bases to effective addresses.

This required some changes in the interfaces between certain parts of the model, e.g., in the signature of the ACL2 functions to read and write memory. Generally, changes to interfaces and their semantics may break proofs whose restoration may require different or more general lemmas and invariants. As explained in Section 2.2, the 64-bit model was accompanied by several correctness proofs of non-trivial 64-bit programs — besides the termination, guard, and other proofs in the model proper. To keep the adaptation of all these proofs more manageable, the changes had to be carried out in incremental steps, to the extent possible. An overarching goal was to keep the whole build working at all times.

3.2 Mode Discrimination

Since the processor does certain things differently depending on whether the current mode is 64-bit or 32-bit, a predicate is needed to discriminate between the two modes in the model. The 64-bit model included a predicate 64-bit-modep over the x86 state, defined to be always t, which was rarely called. This predicate was extended to distinguish between the two modes: if the LMA bit of the IA32_EFER

[3]All such measurements mentioned in this paper have been done on a machine with Intel Xeon E31280 CPU @ 3.50GHz with 32GB RAM.

register is set (i.e., the processor is in IA-32e mode), and the `L` bit of the `CS` register[4] is set (i.e., the current application is running in 64-bit sub-mode, not in compatibility 32-bit sub-mode), the predicate returns `t`; otherwise, the processor is in either protected 32-bit mode or compatibility 32-bit mode, and the predicate returns `nil`. Later, calls of this predicate were replaced by `(eql proc-mode *64-bit-mode*)`, where the value of the variable `proc-mode` is the current processor mode, read once from the x86 state at each execution step and passed to many of the model's functions; for simplicity, in the rest of the paper we still show calls of `64-bit-modep`.

In order to extend the instructions covered by the 64-bit model to 32-bit mode one by one, each branch of the top-level opcode-based dispatch was wrapped into a conditional of this form:

```
(if (64-bit-modep x86) <do-as-before> <throw-error>)
```

This let the model behave as before in 64-bit mode, and provided a clear indication, in concrete or symbolic execution, of instructions not yet extended to 32-bit mode. The addition of these conditionals required the addition of many `64-bit-modep` hypotheses to the existing theorems to keep all the proofs working (which rely on instructions not throwing errors in normal circumstances), a tedious but simple task; however, the extensions described in the subsequent sections required additional work to keep all the proofs working — see Section 3.10 for details.

3.3 Segmentation

Instructions manipulate operands in registers and memory, including stack operands and immediate operands. The register access functions in the 64-bit model actually needed no extension for 32-bit mode. However, as mentioned in Section 3.1, the memory access functions in the 64-bit model had to be extended to take segmentation into account. These memory access functions are used not only for operands, but also for fetching the bytes that form instructions (prefixes, opcodes, etc.). So adding segmentation was a prerequisite to extending any instruction. Segmentation was modeled in a form that is used uniformly in both 64-bit and 32-bit mode, as explained below.

The 64-bit model already included state components for the segment registers, including their hidden parts. A new function `segment-base-and-bound` was added to retrieve the base and bounds of a segment from (the hidden part of) the corresponding segment register in the x86 state. This function takes as arguments the machine state and (the identifying index of) a segment register and returns the base address, lower bound, and upper bounds of the segment as results via `mv`. More precisely, in 64-bit mode, the bases of the `FS` and `GS` segments are retrieved from model-specific registers that are physically mapped (in the processor) to the hidden base fields of the `FS` and `GS` registers, and that provide 64-bit linear addresses beyond the 32-bit linear addresses used in 32-bit mode. The bounds of a segment refer to effective addresses, which are offsets into the segment. The lower bound is 0 and the upper bound is the limit field in the segment register, unless the `E` bit in the segment register is set. This `E` bit indicates an expand-down segment (typically used for a stack): the lower bound is the limit field in the segment register, and the upper bound is either $2^{32}-1$ or $2^{16}-1$, depending on the `D/B` bit in the segment register, which indicates a 32-bit segment if set and a 16-bit segment if cleared. In 64-bit mode, since the bounds are ignored, `segment-base-and-bounds` just returns 0 as both bounds; it also just returns 0 as the base if the segment register is not `FS` or `GS`.

A new function `ea-to-la` was added to perform segment address translation; the name stands for 'effective address to linear address', chosen in analogy to the `la-to-pa` function that, in the 64-bit model,

[4]By expressions like 'bit *x* of the segment register *y*' we mean, more precisely, 'bit *x* of the segment descriptor whose selector is currently loaded in the segment register *y*'.

performed paging address translation, i.e., turned linear addresses into physical addresses.[5] ea-to-la takes as inputs the machine state and a logical address, which consists of an effective address and (the index of) a segment register; it returns as output a linear address, obtained by adding the effective address to the base of the segment. ea-to-la also returns as output a flag with error information if the effective address is outside the segment's bounds; error flag and linear address are returned via mv. The segment base and bounds are retrieved via the function segment-base-and-bounds described earlier. In 64-bit mode, there are no checks against the segment's bounds, but instead the resulting address is checked to be canonical; the resulting address differs from the input effective address only if the segment is FS or GS, whose base is added to the effective address.

The 64-bit model included top-level memory access functions to read and write unsigned and signed values of different sizes (bytes, words, double words, etc.). These were called rmXX, rimXX, wmXX, and wimXX, where r indicates 'read', w indicates 'write', i indicates 'integer' (vs. natural number, i.e., the functions with i read/write signed integers while the functions without i read/write unsigned integers), and XX consists of two digits indicating the bit size (08, 16, 32, etc.). There were also more general functions rm-size, rim-size, wm-size, and wim-size, which took the size in bytes as an additional input. These functions took linear addresses as inputs; they assumed that the base of the FS or GS segment had already been added when applicable, and that alignment checks had already been performed when needed.

New top-level memory access functions were added to read and write unsigned and signed values of different sizes, mirroring the aforementioned ones. They are called rmeXX, rimeXX, wmeXX, wimeXX, rme-size, rime-size, wme-size, and wime-size, where e stands for 'effective address'. These new functions take as inputs logical addresses (as effective addresses and segment registers)[6] instead of linear addresses, call ea-to-la to obtain linear addresses, perform alignment checks on the linear addresses if indicated by some additional inputs to these new functions, and then call the old 64-bit mode top-level memory access functions (which are no longer at the top level in the extended model). For clarity, the old functions were renamed to rmlXX, rimlXX, wmlXX, wimlXX, rml-size, riml-size, wml-size, and wiml-size, where l stands for 'linear address'.

The inputs of all these functions (new and old) include the machine state; the inputs of the write functions also include the value to write, while the inputs of the read functions also include a flag :r or :x to distinguish between reading data and fetching instructions. The read functions return an error flag, the value read, and the possibly updated machine state (see Footnote 9), via mv. The write functions return an error flag and the possibly updated machine state, via mv.

3.4 Paging

As explained in Section 2.1, there are three kinds of paging in the x86 processor. PAE and 32-bit paging are used only in 32-bit mode, and 4-level paging is used only in 64-bit mode. The 64-bit model included the latter, but not the former.[7]

As explained in Section 2.2, the 64-bit model included the specification of paging only in its system-level view, not in its application-level view. These views carry over to the extended model as well.

[5]ea-to-la actually translates a logical address to a linear address, but the name la-to-la would not have worked well. There is a sense in which the effective address portion of a logical address is more "important" than the segment selector, which provides just a "context" for the translation.

[6]A similar observation applies to these names, as the one made for the name of ea-to-la above.

[7]To be precise, a file in the 64-bit model contained a start towards modeling the two paging modes used in 32-bit mode, but this file's content was not used in the rest of the model.

However, the extended model supports 32-bit mode only in application-level view for now, that is, it only supports the execution and verification of 32-bit application software, not 32-bit system software. As such, we have deferred the addition of PAE and 32-bit paging to the model (see Section 5).

3.5 Instruction Fetching

The 64-bit model fetched instruction bytes by using the instruction pointer in the `RIP` register as the linear address passed to the top-level memory access functions `rm08` etc. mentioned in Section 3.3. The instruction pointer was incremented using ACL2's `+` operation, checking that the result was canonical after each increment.

In 32-bit mode, the instruction pointer is in the `EIP` or `IP` register, which refer to the low 32-bit or 16-bit portions of the `RIP` register, respectively. The `D` bit in the `CS` register selects the appropriate register: `EIP` if this bit is set and `IP` otherwise.

A new function `read-*ip` (where the `*` is meant to stand for `r`, `e`, or nothing) was added, to generalize the 64-bit mode reading of the instruction pointer from the `RIP` register. This new function takes as input the machine state and returns as output the instruction pointer. In 32-bit mode, the output is a 32-bit or 16-bit value, from the low bits of the `rip` component of the model of the x86 state.

A new function `add-to-*ip` was added to generalize the plain 64-bit increment of the instruction pointer and the subsequent canonical address check. This new function takes as inputs the machine state, an instruction pointer, and an integer amount to add (which may be negative); it returns as outputs the incremented instruction pointer and an error flag (`nil` if there is no error). In 32-bit mode, the input and output instruction pointers are 32-bit or 16-bit values, and the error flag is non-`nil` if the output instruction pointer is outside the limits of the `CS` register. Note that the input machine state is only used to check the `CS` limits, and that there is no output machine state: this function only increments instruction pointer values.

A new function `write-*ip` was added to store the final instruction pointer, after executing each instruction, into the `RIP`, `EIP`, or `IP` register. This function takes as inputs the machine state and the instruction pointer to store; it returns as output an updated machine state. In 32-bit mode, the instruction pointer is a 32-bit or 16-bit value that is stored in the low 32 or 16 bits of the `rip` component of the model of the x86 state.

The model of instruction fetching was modified to call the new functions `read-*ip` and `add-to-*ip` to manipulate the instruction pointer (while `write-*ip` is called by the models of the individual instructions), and to replace the calls to `rml08` (i.e., the renamed `rm08`; see Section 3.3) with calls to `rme08`. This change necessitated the largest effort, compared to the other changes to the 64-bit model, to keep all the proofs working: see Section 3.10 for details.

3.6 Stack Operations

Several instructions read from and write to the stack, via the stack pointer. The 64-bit model manipulated the stack pointer in a manner similar to the instruction pointer (see Section 3.5): the stack pointer was read from and written to the `RSP` register, and it was incremented and decremented via ACL2's `+` and `-` operations, checking that the result was canonical after each increment and decrement.

In 32-bit mode, the stack pointer is in the `ESP` or `SP` register, which are the low 32-bit or 16-bit portions of the `RSP` register, respectively. The `B` bit in the `SS` register selects the appropriate register: `ESP` if this bit is set and `SP` otherwise.

The extensions to the model for the stack pointer were similar to the extensions for the instruction pointer. New functions read-*sp, add-to-*sp, and write-*sp were added, quite analogous to the functions read-*ip, add-to-*ip, and write-*ip described in Section 3.5. Instead of rip, read-*sp and write-*sp read and write the rsp component of the model of the x86 state — only the low 32 or 16 bits, in 32-bit mode. Instead of the limits of the CS segment, add-to-*sp checks the new stack pointer value against the limits of the SS segment, taking into account the possibility of expand-down segments, which are typically used for the stack.

3.7 Addressing Modes

Other than stack and immediate operands, instructions can reference operands in memory via a variety of addressing modes. Each addressing mode involves a calculation of an effective address from one or more components, such as (the content of) a base register, an index to add to the base register, and a scaling factor with which to multiply the index before adding it to the base. The addressing mode to use is specified by the ModR/M, SIB, and displacement bytes of an instruction (not all of these bytes need to be present). There are 32-bit and 16-bit addressing modes [15, Tables 2.1 and 2.2]: the former apply to 64-bit mode, and to 32-bit mode when the address size is 32 bits; the latter apply to 32-bit mode when the address size is 16 bits. The address size is determined by the D bit of the CS register.

The 64-bit model included functions to decode the ModR/M and SIB bytes and to perform the effective address calculations for the 32-bit addressing modes, but not for the 16-bit addressing modes, which do not apply to 64-bit mode. In particular, the function x86-effective-addr performed the ModR/M and SIB decoding, and the effective address calculation, for the 32-bit addressing modes. Since the 16-bit addressing modes are fairly different from the 32-bit addressing modes, x86-effective-addr was renamed to x86-effective-addr-32/64 (to convey that this is used for 32-bit and 64-bit addresses), a new function x86-effective-addr-16 was added for the 16-bit addressing modes, and a new wrapper function x86-effective-addr was added that calls either x86-effective-addr-32/64 or x86-effective-addr-16.

These functions have several inputs and several outputs. The inputs include the ModR/M and SIB bytes (0 if absent), the current instruction pointer (which is just past the opcode and, if present, the ModR/M and SIB bytes), and the machine state. The outputs are an error flag, the calculated effective address, the number of (displacement) bytes read (which is later added to the instruction pointer), and a possibly updated machine state. Based on the addressing mode, these functions may read the displacement bytes, advancing the instruction pointer; they call rme08 to read these bytes, which may fail (see Section 3.3), in which case x86-effective-addr returns a non-nil error flag.

3.8 Operand Reading and Writing

Some instructions read and write their operands from and to memory or registers, based on some bits of the ModR/M byte — which, as explained in Section 3.7, is also used to determine the addressing mode when the operand is in the memory.

The 64-bit model included a function x86-operand-from-modr/m-and-sib-bytes that uniformly read an operand from memory or registers based on the ModR/M and SIB bytes. This function decoded the ModR/M byte, and, based on that, either read the operand value from a register, or from memory. In the latter case, it called x86-effective-addr to calculate the effective address, which it passed to rml-size (formerly rm-size) or its variants to read the value from that location; recall that in the 64-bit model an effective address was also a linear address. This function had several inputs and several outputs. The inputs included the ModR/M and SIB bytes, the current instruction pointer (which is just past the opcode and, if present, the ModR/M and SIB bytes), and the machine state. The outputs included an error flag, the memory or register operand read, the calculated effective address (for a memory operand), the number of (displacement) bytes read (which is later added to the instruction pointer), and a possibly updated machine state. The calculated effective address is returned to avoid recalculating it when an instruction updates the operand, which is common (e.g., adding an immediate value to an operand in memory).

To uniformly write operands to memory or registers, the 64-bit model also included functions `x86-operand-to-reg/mem` and `x86-operand-to-xmm/mem`. These functions had several inputs, including the value to write, the effective/linear address (calculated externally, often by `x86-operand-from-modr/m-and-sib-bytes` as mentioned above), and the machine state. These functions returned an error flag and an updated machine state. The effective/linear address was passed to `wml-size` (formerly `wm-size`) or its variants to write the operand value to memory.

To support 32-bit mode, these operand read and write functions had to be generalized to traffic in effective addresses rather than linear addresses. In particular, the index of the segment register to use had to be an additional input. This required a change to the interface of these three functions, which are called in many places by the instruction models. Making this change in one shot would have required a lot of adjustments to instruction models and could have broken many proofs. Thus, the change was staged as follows: (1) introduce new versions of these functions, called `x86-operand-from-modr/m-and-sib-bytes$`, `x86-operand-to-reg/mem$`, and `x86-operand-to-xmm/mem$`, with the new interface and functionality; (2) separately modify the instruction models, one by one, to call the new functions instead of the old ones, repairing any failing proofs; (3) remove the old functions when they are no longer called; (4) rename the new functions to remove the ending `$`. The new functions to read and write operands call the new top-level memory functions `rme-size` and `wme-size` or their variants to access the operands in memory.

The segment register to pass to the new functions is determined using the default segment selection rules [15, Volume 1, Table 3-5] based on the kind of instruction and the addressing mode, and the presence of a segment-override prefix before the opcode. This determination is made by a new function `select-segment-register` that was added for this purpose. This function has several inputs, including the segment override prefix (0 if absent) and the machine state; this functions returns (the index of) a segment register as the only output.

3.9 Instructions

With all the extensions described above in place, extending the individual instructions' semantic functions to 32-bit mode was comparatively easy. This was also facilitated by the fact that the 64-bit model already supported different operand sizes for many of the core operations carried out by the instructions (e.g., the arithmetic and logic operations) leaving just the operand size determination logic of the instruction to be extended.

In particular, the existing function `select-operand-size` was extended to return the operand size in 32-bit mode, while returning the same result as before in 64-bit mode. The interface of this function was expanded to give it access to the `D` bit of the `CS` register, needed to determine the operand size in 32-bit mode: its inputs include the machine state and some of the decoded instruction prefixes; it returns the size as the only output. The instruction semantic functions that already called `select-operand-size` could then automatically extend their operand size determination to 32-bit mode.

A new function `select-address-size` was introduced to determine the address size in both 32-bit and 64-bit mode. Its inputs include the machine state and some of the decoded instruction prefixes; it returns the size as the only output. The portions of the instruction semantic functions that determined the address size in 64-bit mode only were replaced with calls to this new function.

In the 64-bit model, the instruction semantic functions were reading immediate operands via `rm08` and related functions (renamed to `rml08` etc.) called on the instruction pointer (treated as a linear address), incrementing the instruction pointer via ACL2's `+`, and writing the final instruction pointer directly into the `rip` component of the x86 state. To support 32-bit mode as well, this was changed to read immediate operands via `rme08` and related functions called on the instruction pointer (treated as an effective address in the code segment), and to use the new functions described in Section 3.5 to increment and store the instruction pointer.

In the 64-bit model, the instruction semantic functions were reading and writing the stack pointer directly from and to the `rsp` component of the x86 state, calling `rm-size`, `wm-size`, and related functions (renamed to `rml-size`, `wml-size`, etc.) on the stack pointer (treated as a linear address) to read and write stack operands, and incrementing and decrementing the stack pointer via ACL2's `+` and `-`. To support 32-bit mode as well, this was changed to call `rme-size`, `wme-size`, and related functions on the stack pointer (treated as an effective address in the stack segment) to read and write stack operands, and to use the new functions described in Section 3.6 to read, write, increment, and decrement the stack pointer. The instruction semantic functions in the 64-bit model were performing alignment checks on the stack pointer as needed: this code was removed because alignment checks are performed inside `rme-size`, `wme-size`, and related functions, after effective addresses are translated to linear addresses, resulting in better factored code as a by-product.

Following the staged approach described in Section 3.8, calls to `x86-operand-from-modr/m-and-sib-bytes` etc. in the instruction semantic functions, were gradually redirected to call `x86-operand-from-modr/m-and-sib-bytes$` etc. instead. Since these new functions traffic in effective addresses instead of linear addresses, two additional adjustments had to be made to the instruction semantic functions. The first adjustment was to remove alignment checks performed on the linear addresses of the operands: alignment checks are already performed by (functions called by) the new functions, after translating the now effective addresses of the operands into linear addresses. The second adjustment was to remove the addition of the `FS` or `GS` segment base (when applicable) to the linear addresses of the operands: any segment base is added to the now effective addresses of the operands when they are translated into linear addresses, by (functions called by) the new functions. These adjustments resulted in better factored code as a by-product.

After each instruction semantic function was extended to 32-bit mode in the manner explained above, the top-level dispatch was adjusted to call the function not only in 64-bit mode, but also in 32-bit mode. Concretely, this was done by removing the wrapping conditional `(if (64-bit-modep x86) ...)` described in Section 3.2.

All the non-floating-point instructions of the 64-bit model have been extended to 32-bit mode, at least for the application-level view of execution.[8] The 34 floating-point instructions supported by the 64-bit model also need to be extended to 32-bit mode. We anticipate that this task will be straightforward because we have almost all the pieces necessary to extend the floating-point instructions. Specification functions from Russinoff's RTL library [3] are used to specify the core operations of these instructions, and these functions are already parameterized by size. As far as operand access and update is concerned: we have already extended memory access functions to work in 32-bit mode, but since the floating-point instructions use a different register set from the general-purpose instructions, we just need to extend

[8]The far variant of `JMP` does not handle 32-bit mode system segments yet, but these are only needed for the system-level view of execution. Nonetheless, we plan to add support for these soon.

register access functions to work in 32-bit mode.

3.10 Proof Adaptations

In extending the 64-bit ISA model to support 32-bit mode, so far we have discussed issues that were not especially unique to a formal model; for instance, extending a C emulator of the 64-bit x86 ISA to support the execution of 32-bit programs would likely require making similar changes. However, as discussed in Section 3.1, our aim was also to preserve the proofs done in the 64-bit model, including the proofs of the model's functions (e.g., guards) and the proofs of correctness of the 64-bit programs.

The proofs of 64-bit programs with the 64-bit model involved functions and expressions that were generalized for 32-bit mode as described in the previous subsections. Thus, theorems were added to rewrite the more general functions and expressions into the "old" ones under the `64-bit-modep` hypothesis. For instance, the following theorem was added, to rewrite the reading of the `RSP`, `ESP`, or `SP` register into just the reading of `RSP` (where `rgfi` reads the 64-bit value of the register specified as argument):

```
(implies (64-bit-modep x86)
         (equal (read-*sp x86) (rgfi *rsp* x86)))
```

Since the 64-bit proofs include the `64-bit-modep` hypothesis (added as described in Section 3.2), these rewrite rules fire and help reduce (sub)goals to the form they had in the 64-bit model prior to its extensions. Additional theorems had to be added in order to let the `64-bit-modep` hypothesis be relieved for updated x86 states, such as the following one, asserting that reading from a linear address does not change mode:[9]

```
(equal (64-bit-modep (mv-nth 2 (rml08 addr r-x x86)))
       (64-bit-modep x86))
```

All these theorems generally sufficed to keep the proofs of 64-bit application programs, i.e., in the application-level view.

Even though the extended model does not (yet) cover the system-level view, additional changes were needed to adapt the reasoning strategy to work for the proofs of some 64-bit system programs, as described below. Lemma libraries included in the 64-bit model for formally analyzing system programs contained utilities to reason about the memory management data structures — specifically, the paging structures. Every linear-to-physical address translation on the x86 ISA causes a traversal of entries in these paging structures,[10] which produces some side effects. The processor can set two bits in the paging entries during address translation: the *accessed* and *dirty* flags, which effectively *mark* the entries that govern the translation of a linear address. However, the mapping of that linear address to its corresponding physical address does not change as a result of these side-effect updates. This means that statements like the following are true, where `spec-function` is an x86 specification function that reads from or writes to memory, and <hyp> ensures that (1) `x86-1` and `x86-2` can differ only in the values of the accessed and dirty flags, and (2) `spec-function` traffics only in locations that are disjoint from the affected (i.e., marked) entries in the paging structures:

```
(implies <hyp> (equal (spec-function <args> x86-1) (spec-function <args> x86-2)))
```

The 64-bit model stored such theorems as congruence rules, as follows:

[9]Note that reading from memory may change the x86 state in general, by changing the 'accessed' flag in the paging data structures; this flag is discussed later in this section. `rml08` and similar functions return the possibly updated state as the third component of their multi-value result, accessed via `(mv-nth 2 ...)`.

[10]We ignore translation look-aside buffers and caches for this discussion.

```
(implies (xlate-equiv-memory x86-1 x86-2)
         (equal (spec-function-alt <args> x86-1)
                (spec-function-alt <args> x86-2)))
```

The function `spec-function-alt` is exactly the same as `spec-function` under one condition: it traffics only in locations that are disjoint from all the paging entries that are marked during address translation; if this condition is false, then `spec-function-alt` simply returns the input x86 state — unmodified. The relation `xlate-equiv-memory` says that two x86 states are equivalent if: (1) their paging structures are equal, modulo the accessed and dirty bits; and (2) all other memory locations are equal. This arrangement facilitates "conditional" congruence-based reasoning[11] — a rewrite rule transforms `spec-function` to `spec-function-alt` whenever applicable, and then these congruence rules can come into play. This strategy works well for system programs that do not explicitly modify the paging data structures. As a result of extending the model, we had to add `64-bit-modep` both to `xlate-equiv-memory` and to the conditions under which `spec-function-alt` is equal to `spec-function`.

We note that the top-level rewrite rules in the model are not mode-specific — an example is the opener lemma of the step function, whose hypotheses determine when to rewrite a call of the step function to the function that dispatches control to the appropriate instruction semantic function; basically, this lemma helps in "unwinding" the x86 interpreter during proofs. Thus, we share lemmas across different modes as much as possible. Rules about intermediate specification functions that do have a hypothesis like `(64-bit-modep x86)` come in useful during program verification when we need to be cognizant of the operating mode of a processor.

3.11 Performance

As discussed in Section 2.2, a high simulation speed is crucial for performing model validation via co-simulations against real machines. We first discuss how the simulation speed of 64-bit programs was adversely impacted by extending the model to support 32-bit programs and how we fixed this issue. Then we present performance-related information about the simulation of 32-bit programs.

3.11.1 64-bit Mode

Our initial attempt to extend the 64-bit model led to an unfortunate decrease in its simulation speed in the application-level view from around 3 million instructions/second to 1.9 million instructions/second.[12] Using Linux's Perf utility and ACL2's `profile` mechanism [1, `:doc profile`], we discovered that the predicate `64-bit-modep` was the main culprit — every call of this function allocated 16 bytes on the heap. Recall that this predicate reads values from fields in the x86 state (`IA32_EFER` and `CS` registers) and these fields usually contain large integers, i.e., *bignums*. Manipulating bignums is inefficient because they are stored on the heap and require special arithmetic functions. Contrast these bignums with *fixnum* integers, which can fit in the host machine's registers and whose computation can be done with built-in arithmetic instructions. For example, Lisp compilers can add two fixnum values using the add instruction of the host machine, but the addition of two bignum values is done by calling a Lisp function that is considerably slower than a machine instruction and that may also allocate memory on the heap.

Though `64-bit-modep` allocates just 16 bytes on the heap, it is called once at the beginning of the execution of *each* instruction to determine the processor's operating mode for that instruction. Therefore,

[11] Recall that a congruence rule cannot have anything other than a known equivalence relation as its only hypothesis.

[12] There was a similar decrease in speed in the system-level view, but for this discussion, we focus only on the application-level view.

the number of bytes allocated because of this function increases with the number of instructions to be executed. For instance, for a simple C function that computes the `n`-th Fibonacci number by implementing the Fibonacci recurrence relation, the number of 64-bit x86 instructions to be executed to compute `fib(30)` was 31,281,993. This program run allocated 500,511,936 bytes on the model, all because of the bignum computations done in `64-bit-modep`. Since we typically run a large number of instructions at once during co-simulations, this problem needed to be fixed.

We solved this issue by changing the definition of `64-bit-modep` — it is now an `mbe`, where our initial definition has been retained in the `:logic` part. For the `:exec` part, we used a function called `bignum-extract` from a pre-existing ACL2 community book [1, `:doc bignum-extract`]. Logically, `(bignum-extract x n)` returns the `n`-th 32-bit slice of an integer `x`. Therefore, in the `:exec` part of `64-bit-modep`, we simply extracted relevant 32-bit slices of the x86 fields using this function and read the required values from them. During execution, we used a raw Lisp replacement for `bignum-extract` (already provided in `std/bitsets/bignum-extract-opt.lisp`) that takes advantage of 64-bit CCL's implementation. CCL stores bignums in vectors of 32 bits, and extracting a 32-bit chunk of a bignum to operate on avoids expensive computations involving the *entire* bignum.

With this solution, the model's simulation speed for 64-bit programs went back to what it was at the beginning of this project (for both application-level and system-level views). We note that the number of bytes allocated on the heap to compute `fib(n)` is now independent of the number of instructions to be executed.

However, this solution does have two drawbacks:

1. The raw Lisp replacement for `bignum-extract` may not be efficient for Lisps other than CCL.
2. For efficient execution (not for reasoning), this solution requires a trust tag.

In the future, we plan to implement a different solution that will not suffer from the above drawbacks. We will take advantage of the x86 abstract stobj by defining the fields involved in the computation of `64-bit-modep` as multiple fixnum-sized fields in the concrete stobj, defining accessor and updater functions over the concrete stobj that operate on these fields collectively, and then proving that these concrete functions correspond to the currently existing accessor and updater functions defined over the abstract stobj. A benefit of this approach is that after the abstract stobj is defined, no other proofs would be affected.

3.11.2 32-bit Mode

The simulation speed of 32-bit programs in the application-level view is around 920,000 instructions/ second — note that, for 32-bit mode, the system-level view has not been implemented yet. One reason for this slower speed (as compared to 64-bit mode) is that 32-bit programs often have to refer to segment descriptors over the course of their execution, and these operations usually involve bignums.

Our focus has been on the functionality of the extensions first rather than the performance, but in the future, we intend to speed up 32-bit mode using the same kind of approaches as those used in 64-bit mode. That being said, the current simulation speed of 32-bit programs has been more than adequate to run tests at this stage of development.

3.12 Documentation

As mentioned in Section 2.2, the 64-bit model was extensively documented. As the model was being extended to support 32-bit mode, the documentation was extended accordingly.

4 Related Work

The x86-CC and x86-TSO models of the x86 ISA in HOL [23, 25] are focused on concurrent memory access in multiprocessor systems. They support a few tens of instructions, defined via parameterized monadic “micro-code” operations that can be instantiated for both sequential and concurrent semantics. The models are accompanied by a tool to test the sequential semantics in HOL, and a tool to test the concurrent semantics in (unverified) OCaml.

The model of the x86 ISA in Coq developed for RockSalt [20] supports the verification of a binary analyzer that checks conformance to a sandboxing policy; however, the model is independent of this analyzer. It supports about 100 instructions in 32-bit mode only, and does not model paging tables. It uses two Domain-Specific Languages (DSLs) embedded in Coq to declaratively specify instruction decoding and to express instruction semantics in terms of “micro-code” operations. The model can simulate and validate (against real processors) about 50 instructions per second.

Degenbaev’s model of the x86 ISA [9] supports both concurrent memory access in multiprocessor systems and a large number of instructions; in this work, the instruction semantics is modeled via a DSL. Baumann’s related model of the x87 instruction set [6] supports floating-point instructions. These models are written in “pencil-and-paper”, not in a theorem prover or other formal tool.

Models of the ISA of other processors have been developed in ACL2 and other theorem provers, e.g., [24, 10, 22] In particular, the model of the y86 ISA (an x86-like ISA) in ACL2 [4] is a precursor of our model. The modeling approaches for these other processors are similar to the ones for the x86 ISA models in ACL2 and other provers, but the ISAs of these other processors are less complicated than the x86.

Models of both intermediate-level and source-level languages have been also developed in ACL2 and other theorem provers, e.g., [17, 21]. There are some commonalities between the modeling approaches for these languages and the modeling approaches for processor ISAs, such as the suitability of an interpreter style of operational semantics. There are also many differences, e.g., processor ISAs typically have fewer data types and do not have to include specifications of operations like casting (e.g., converting integers to characters, using pointers as integers, etc.).

There exist several x86 emulators (e.g., Bochs, DOSBox, PCem, QEMU, Unicorn) that are written in more conventional programming languages (e.g., C++), not in the logical languages of theorem provers. The development of these emulators faces many of the same issues as the development of a formal model, such as the sheer complexity and the occasional ambiguity of the official documentation. However, the proof aspects are unique to formal models. In particular, extending or improving an emulator involves regression testing, while extending or improving a formal model may also involve proof adaptation.

5 Future Work

In this paper, we discussed our ongoing effort to extend the pre-existing 64-bit model of the x86 ISA to support Intel’s 32-bit mode. In the short or medium term, we plan to add support for the following x86 ISA features:

- More instructions, prioritizing on the ones encountered in new programs to verify and on the ones that are more commonly used in general. Every new instruction will include both 64-bit and 32-bit mode support.
- The two kinds of paging used in 32-bit mode (see Section 3.4). This will enable the formal analysis of 32-bit system software, besides 32-bit application software.

- The remaining processor modes (real-address, virtual-8086, and system management), along with the instructions to switch between modes.
- The widely-used Intel's vector processing features (i.e., AVX, AVX2, AVX-512) that offer enhanced capabilities to perform data-intensive (integer and floating-point) computations efficiently. The x86 model already supports the decoding of these instructions, but this task will entail modifying some old instructions that now support these new features and adding many new instructions. A notable thing is that vector processing features differ between the 64-bit and 32-bit modes, thereby making this task a little more interesting.

In the short or medium term, we also plan to validate the 32-bit extensions against actual x86 processors, as was done for the 64-bit model.

A longer-term project is to support a concurrent semantics. At the very least, this will likely require some refactoring of the instruction semantic functions, which currently treat the execution of each instruction as an atomic event, because the x86 state is updated at the end of each instruction.

Another direction is to make some parts of the specification more declarative and concise via more macros to generate boilerplate code. Examples are the binding of various instruction bytes and fields (e.g., prefixes) to variables and common checks (e.g., many instructions throw a `#UD` exception if the `LOCK` prefix is used). A more ambitious endeavor is to raise the level of abstraction of some parts of the specification to the point of making them non-executable, and then use transformation tools like APT [5, 8] to generate efficient executable refinements of the specification along with correctness proofs checked by ACL2.

As briefly mentioned in Section 1, some work has started on 32-bit program verification, mainly aimed at detecting malware variants via semantic equivalence checking by symbolic execution. This work, performed by Eric Smith with some help about the model from the first author, will be carried forward in the near future.

Longer-term envisioned uses of the model include verified compilation from programming languages (e.g. C) to binaries, as well as binary program derivations by stepwise refinement using transformation tools like APT [5, 8], possibly on deeply embedded representations [7].

Acknowledgements

This work was partially supported by DARPA under Contract No. D16PC00178. Thanks to Eric Smith and James McDonald for providing feedback based on their use of the model for proofs, symbolic execution, and simulation.

References

[1] *ACL2 Theorem Prover and Community Books: User Manual*. Online; accessed: September 2018. `http://www.cs.utexas.edu/~moore/acl2/manuals/current/manual`.

[2] *ACL2 Books: Codewalker*. Online; accessed: September 2018. `https://github.com/acl2/acl2/tree/master/books/projects/codewalker`.

[3] *ACL2 Books:* `rtl/rel11` *- A Formal Theory of Register-Transfer Logic and Computer Arithmetic*. Online; accessed: September 2018. `https://github.com/acl2/acl2/tree/master/books/rtl/rel11`.

[4] *ACL2 Books: y86 Specifications*. Online; accessed: September 2018. `https://github.com/acl2/acl2/tree/master/books/models/y86`.

[5] *APT (Automated Program Transformations): Web page*. Online; accessed: September 2018.
`http://www.kestrel.edu/home/projects/apt`.

[6] Christoph Baumann (2008): *Formal Specification of the x86 Floating-Point Instruction Set*. Master's thesis, Universität des Saarlandes. Online; accessed: September 2018.
`http://www-wjp.cs.uni-saarland.de/publikationen/Ba08.pdf`.

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

[8] Alessandro Coglio, Matt Kaufmann & Eric Smith (2017): *A Versatile, Sound Tool for Simplifying Definitions*. In: Proc. 14th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2017), pp. 61–77, doi:10.4204/EPTCS.249.5.

[9] Ulan Degenbaev (2012): *Formal Specification of the x86 Instruction Set Architecture*. Ph.D. thesis, Universität des Saarlandes. Online; accessed: September 2018.
`http://rg-master.cs.uni-sb.de/publikationen/UD11.pdf`.

[10] Anthony Fox & Magnus O. Myreen (2010): *A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture*. In Matt Kaufmann and Lawrence C. Paulson, editor: Interactive Theorem Proving, Lecture Notes in Computer Science 6172, Springer Berlin Heidelberg, pp. 243–258, doi:10.1007/978-3-642-14052-5_18.

[11] 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. Online; accessed: September 2018.
`http://hdl.handle.net/2152/46437`.

[12] Shilpi Goel (2017): *The x86isa Books: Features, Usage, and Future Plans*. In: Proc. 14th International Workshop on the ACL2 Theorem Prover and its Applications (ACL2-2017), pp. 1–17, doi:10.4204/EPTCS.249.1.

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

[14] Shilpi Goel, Warren A. Hunt, Jr. & Matt Kaufmann (2017): *Engineering a Formal, Executable x86 ISA Simulator for Software Verification*. In: Provably Correct Systems, Springer International Publishing, Cham, pp. 173–209, doi:10.1007/978-3-319-48628-4_8. Editors: Mike Hinchey, Jonathan P. Bowen, and Ernst-Rüdiger.

[15] Intel Corporation: *Intel 64 and IA-32 Architectures Software Developer's Manual*. Online; accessed: September 2018. Order Number: 325462-067US. (May, 2018).
`http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`.

[16] John Leyden (2017): *64-bit malware threat may be itty-bitty now, but it's only set to grow*. The Register. Online; accessed: September 2018.
`https://www.theregister.co.uk/2017/05/24/64bit_malware`.

[17] Hanbing Liu & J S. Moore (2004): *Java program verification via a JVM deep embedding in ACL2*. In: International Conference on Theorem Proving in Higher Order Logics, Springer, pp. 184–200, doi:10.1007/978-3-540-30142-4_14.

[18] John McCarthy (1964): *A Formal Description of a Subset of Algol*. In T. B. Steel, editor: Formal Language Description Languages for Computer Programming, North Holland, 1966, pp. 1–12.

[19] J S. Moore: *Mechanized Operational Semantics*. Online; accessed: September 2018. Lectures in the Marktoberdorf Summer School (August 5-16, 2008).
`http://www.cs.utexas.edu/users/moore/publications/talks/marktoberdorf-08/index.html`.

[20] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan & Edward Gan (2012): *RockSalt: Better, Faster, Stronger SFI for the x86*. In: Proc. of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, ACM, pp. 395–404, doi:10.1145/2254064.2254111.

[21] Michael Norrish (1998): *C formalised in HOL*. Ph.D. thesis, University of Cambridge. Online; accessed: September 2018.
`https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-453.pdf`.

[22] Alastair Reid (2016): *Trustworthy specifications of ARM® v8-A and v8-M System level architecture*. In: *2016 Formal Methods in Computer-Aided Design (FMCAD)*, pp. 161–168, doi:10.1109/FMCAD.2016.7886675.

[23] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus Myreen & Jade Alglave (2009): *The Semantics of x86-CC Multiprocessor Machine Code*. In: *Proc. 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 379–391, doi:10.1145/1594834.1480929.

[24] Jun Sawada & Warren A. Hunt, Jr. (2002): *Verification of FM9801: An Out-of-Order Microprocessor Model with Speculative Execution, Exceptions, and Program-Modifying Capability*. *Formal Methods in Systems Design* 20(2), pp. 187–222, doi:10.1023/A:1014122630277.

[25] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli & Magnus O. Myreen (2010): *x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors*. *Communications of the ACM* 53(7), pp. 89–97, doi:10.1145/1785414.1785443.

[26] 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. Online; accessed: September 2018.
`http://hdl.handle.net/2152/ETD-UT-2010-12-2210`.

[27] Sol Swords & Jared Davis (2011): *Bit-Blasting ACL2 Theorems*. In: *Proc. 10th International Workshop on the ACL2 Theorem Prover and its Applications (ACL2-2011)*, pp. 84–102, doi:10.4204/EPTCS.70.7.

A Toolbox For Property Checking From Simulation Using Incremental SAT (Extended Abstract)

Rob Sumners
Centaur Technology
`rsumners@centtech.com`

We present a tool that primarily supports the ability to check bounded properties starting from a sequence of states in a run. The target design is compiled into an AIGNET which is then selectively and iteratively translated into an incremental SAT instance in which clauses are added for new terms and simplified by the assignment of existing literals. Additional applications of the tool can be derived by the user providing alternative attachments of constrained functions which guide the iterations and SAT checks performed. Some Verilog RTL examples are included for reference.

1 Overview

Formal property verification of modern hardware systems is often too onerous to prove directly with a theorem prover and too expensive to prove with more automatic techniques. A common remedy to this difficulty is to focus on proving properties from a more limited logic (e.g. STE [9], BMC [2], using SVTVs [10] in previous ACL2 work). In these cases, properties are reduced in scope to only consider the effects of a bounded number of next-steps of the transition function of the hardware design; this, in turn, greatly increases the capacity of the hardware designs one can target with automatic propositional logic verification techniques (primarily SAT). The general downside of this reduction in property scope is the need to codify and prove (inductive) invariants of the reachable states of the design. The definition and proof of these invariants is not only time consuming but the invariants are often brittle and require significant maintenance with ongoing design change. An alternative approach is to bypass the definition and proof of invariants and instead check the properties from a set of representative states of the system – one can then either prove that these states are in fact representative or accept the results of the property checks as a sufficient semi-formal check of the design and disregard a full proof. We briefly cover the design and definition of a tool (named `exsim`) which builds upon existing hardware verification work in the ACL2 community (VL [10], SV [10], AIGNET [3]) to add the capability to efficiently check bounded properties across a sequence of next-states of the design using incremental SAT [13]. An initial version of the tool and some example Verilog designs and runs are included in the supporting materials.

2 Design and Toy Example

The `exsim` function is split into two steps. The first step is the function `exsim-prep` which reads in a verilog design and compiles it into an AIGNET – an optimized "circuit" representation built from `AND` and `XOR` gates and inverters supporting efficient simplification procedures [3].

The second step of `exsim` is the function `exsim-run` which reads in a waveform for the design (from a VCD file) and initializes the data structures needed to iterate through the `exsim-main-loop`. As part of this initialization, inputs of the design are tagged as either `:wave`, `:rand`, or `:free` corresponding to

```
module toy (input            reset,            module top (input       reset,
            input            clk,                          input       clk,
            input            op,                           input       wave_op,
            input [1:0]      in,                           input [1:0] free_in,
            output reg [1:0] out);                         output      fail_out);
  ...                                            ...
  always@* w1 = tmp|in;                          always@(posedge clk) in <= free_in;
  always@* w2 = tmp&{2{in[0]}};                  always@(posedge clk) op <= wave_op;
  always@(posedge clk) tmp <= in;                toy des(.*);
  always@(posedge clk) out <= (op ? w1 : w2);    always@(posedge clk) fail_out <= &out;
endmodule // toy                               endmodule // top
```

Figure 1: Definition of a simple `toy` module and `top` environment

input values tracking values from the input waveform, being generated randomly, or being left as free variables. In addition, certain signals are tagged as `:fail` which merely designates these signals as the properties we are checking – namely that these fail signals never have a non-zero value after reset. A simple toy example in Figure 1 is provided for reference – the default signal tagging function would tag `free_in` as the sole `:free` input and `fail_out` as the sole `:fail` signal to check.

The primary function of the `exsim-main-loop` is to update an incremental SAT instance [1] such that the clauses in the SAT instance correspond to a check of `:fail` signals at certain clock cycles having the value of 1 relative to certain values of `:free` inputs on previous clock cycles. The main-loop iterates through a sequence of operations primarily comprised of `:step-fail`, `:step-free`, and `:check-fails` which respectively add clauses to SAT for the :fail at the next clock cycle, add unit clauses to SAT corresponding to an assignment to :free variables on an earlier clock cycle, and calling the SAT solver to check if any of the :fail signals could be 1 in the current context. The "compiled" AIGNET from `exsim-prep` allows for an efficient implementation of `:step-free` and `:step-fail` for larger designs.

The choice of which step to take is heuristic and depends primarily on the current number of clauses in the database. In particular, `:step-fail` will add clauses to SAT while `:step-free` will reduce clauses in SAT and heuristics will choose when to perform each action depending on how extensively the user wishes to search for fails in a given instance. The default heuristic choice function (as well as several other functions defining default operation) can be overridden using ACL2's `defattach` feature [8].

Returning to our simple toy example from Figure 1. Let's assume that the input `wave_op` gets a value of 1 at every clock cycle from the input waves. After a sufficient number of `:step-fails`, this would reduce the check of `fail_out` being 1 to `free_in` having at least a single 1 in each bit position over 2 consecutive clocks, 3 clocks earlier.

The intent of the `exsim` toolkit is to provide the user the capability to control the SAT checks that are performed to optimize effective search capacity over runtime. This is primarily achieved by allowing the user to carefully specify which input variables should be :free (which reduces potential decision space during SAT along with increasing propagations) and when to either extend or reduce the current set of SAT clauses by inspecting the current number of active clauses. The additional benefit of incremental SAT is that the aggregate cost of successive searches is reduced by the sharing of learned reductions.

The `exsim` toolkit is a work in progress. Current development efforts afford better control over bindings of free variables and more intermediate targets for search (not just `:fail` signals). Related to these efforts, an earlier proof of "correctness" needs to be reworked to keep apace with the design changes. As a note on this "proof", the goal is to ensure that each `:check-fails` which returns UNSAT ensures no possible assignment for the free variables in the scope of the current check. This reduces to proving an invariant relating the current SAT clauses with the values of signals at certain clock cycles.

References

[1] Tomás Balyo, Armin Biere, Markus Iser & Carsten Sinz (2016): *SAT Race 2015*. Artif. Intell. 241, pp. 45–65, doi:10.1016/j.artint.2016.08.007.

[2] Edmund M. Clarke, Armin Biere, Richard Raimi & Yunshan Zhu (2001): *Bounded Model Checking Using Satisfiability Solving*. Formal Methods in System Design 19(1), pp. 7–34, doi:10.1023/A:1011276507260.

[3] Jared Davis & Sol Swords (2013): *Verified AIG Algorithms in ACL2*. In Gamboa & Davis [4], pp. 95–110, doi:10.4204/EPTCS.114.8.

[4] Ruben Gamboa & Jared Davis, editors (2013): *Proceedings International Workshop on the ACL2 Theorem Prover and its Applications, ACL2 2013, Laramie, Wyoming, USA, May 30-31, 2013*. EPTCS 114, doi:10.4204/EPTCS.114.

[5] 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.

[6] David Hardin & Julien Schmaltz, editors (2011): *Proceedings 10th International Workshop on the ACL2 Theorem Prover and its Applications, ACL2 2011, Austin, Texas, USA, November 3-4, 2011*. EPTCS 70, doi:10.4204/EPTCS.70.

[7] Warren A. Hunt, Sol Swords, Jared Davis & Anna Slobodova (2010): *Use of Formal Verification at Centaur Technology*, pp. 65–88. Springer US, Boston, MA, doi:10.1007/978-1-4419-1539-9_3.

[8] Matt Kaufmann & J Strother Moore (2014): *Enhancements to ACL2 in Versions 6.2, 6.3, and 6.4*. In Verbeek & Schmaltz [12], pp. 1–7, doi:10.4204/EPTCS.152.1.

[9] Manish Pandey & Randal E. Bryant (1999): *Exploiting symmetry when verifying transistor-level circuits by symbolic trajectory evaluation*. IEEE Trans. on CAD of Integrated Circuits and Systems 18(7), pp. 918–935, doi:10.1109/43.771176.

[10] Anna Slobodová, Jared Davis, Sol Swords & 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, Cambridge, UK, 11-13 July, 2011*, pp. 89–97, doi:10.1109/MEMCOD.2011.5970515.

[11] Sol Swords & Jared Davis (2011): *Bit-Blasting ACL2 Theorems*. In Hardin & Schmaltz [6], pp. 84–102, doi:10.4204/EPTCS.70.7.

[12] Freek Verbeek & Julien Schmaltz, editors (2014): *Proceedings Twelfth International Workshop on the ACL2 Theorem Prover and its Applications, Vienna, Austria, 12-13th July 2014*. EPTCS 152, doi:10.4204/EPTCS.152.

[13] Jesse Whittemore, Joonyoung Kim & Karem A. Sakallah (2001): *SATIRE: A New Incremental Satisfiability Engine*. In: *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pp. 542–545, doi:10.1145/378239.379019.

The Fundamental Theorem of Algebra in ACL2

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

We report on a verification of the Fundamental Theorem of Algebra in ACL2(r). The proof consists of four parts. First, continuity for both complex-valued and real-valued functions of complex numbers is defined, and it is shown that continuous functions from the complex to the real numbers achieve a minimum value over a closed square region. An important case of continuous real-valued, complex functions results from taking the traditional complex norm of a continuous complex function. We think of these continuous functions as having only one (complex) argument, but in ACL2(r) they appear as functions of two arguments. The extra argument is a "context", which is uninterpreted. For example, it could be other arguments that are held fixed, as in an exponential function which has a base and an exponent, either of which could be held fixed. Second, it is shown that complex polynomials are continuous, so the norm of a complex polynomial is a continuous real-valued function and it achieves its minimum over an arbitrary square region centered at the origin. This part of the proof benefits from the introduction of the "context" argument, and it illustrates an innovation that simplifies the proofs of classical properties with unbound parameters. Third, we derive lower and upper bounds on the norm of non-constant polynomials for inputs that are sufficiently far away from the origin. This means that a sufficiently large square can be found to guarantee that it contains the global minimum of the norm of the polynomial. Fourth, it is shown that if a given number is not a root of a non-constant polynomial, then it cannot be the global minimum. Finally, these results are combined to show that the global minimum must be a root of the polynomial. This result is part of a larger effort in the formalization of complex polynomials in ACL2(r).

1 Introduction

In this paper, we describe a verification of the Fundamental Theorem of Algebra (FTA) in ACL2(r). That is, we prove that if p is a non-constant, complex[1] polynomial with complex coefficients, then there is some complex number z such that $p(z) = 0$. The proof follows the outline of the first proof in [3], and it is a formal version of d'Alembert's proof of 1746 [1], which is illustrated (literally) in [11].

Figure 1 provides an outline of the proof, which comprises three main strands. First (step 1 in Figure 1), we show that continuous functions from the complex plane to the reals always achieve a minimum value in a closed, bounded, rectangular region. The consequence to the FTA is that if p is a complex polynomial, then the function mapping $z \in \mathbb{C}$ to $||p(z)||$, where $||\cdot||$ denotes the traditional complex norm, achieves a minimum value in a square region centered at the origin (steps 2, 3, and 5). Second (step 4), we show that the norm of polynomials is dominated by the term of highest power. In particular, if $p(z) = a_n z^n + a_{n-1}z^{n-1} + \cdots + a_0$, then $\frac{1}{2}||a_n||\,||z||^n < ||p(z)|| < \frac{3}{2}||a_n||\,||z||^n$ for sufficiently large z. These two facts can be combined to show that the global minimum of a polynomial (norm) must be enclosed in a possibly large region around the origin, so the polynomial must achieve this global minimum. The third strand (step 6), known as d'Alembert's Lemma, states that if p is a complex

[1] By "complex" we mean the traditional mathematical view that the complex numbers are an extension of the real numbers, not the ACL2 view that complex numbers are necessarily not real.

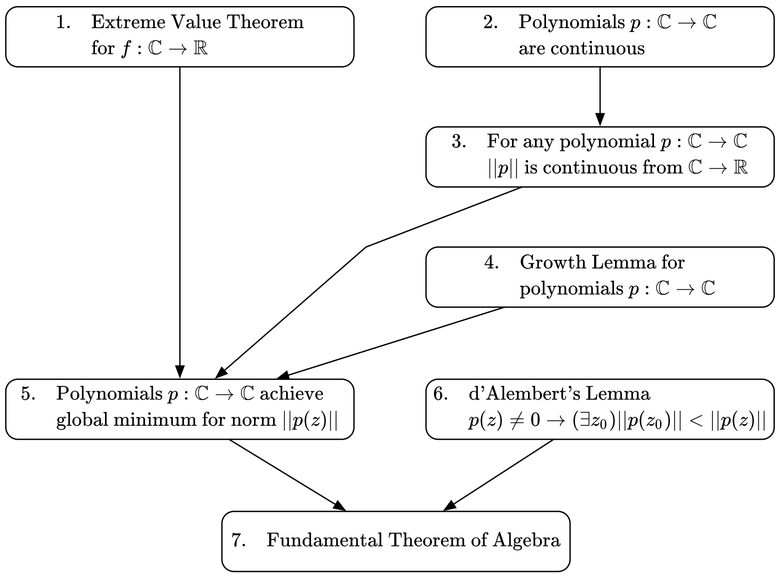

1. Extreme Value Theorem for $f : \mathbb{C} \to \mathbb{R}$
2. Polynomials $p : \mathbb{C} \to \mathbb{C}$ are continuous
3. For any polynomial $p : \mathbb{C} \to \mathbb{C}$ $||p||$ is continuous from $\mathbb{C} \to \mathbb{R}$
4. Growth Lemma for polynomials $p : \mathbb{C} \to \mathbb{C}$
5. Polynomials $p : \mathbb{C} \to \mathbb{C}$ achieve global minimum for norm $||p(z)||$
6. d'Alembert's Lemma $p(z) \neq 0 \to (\exists z_0)||p(z_0)|| < ||p(z)||$
7. Fundamental Theorem of Algebra

Figure 1: Proof Outline

polynomial and z is such that $p(z) \neq 0$, then there is some z_0 such that $||p(z_0)|| < ||p(z)||$. This implies that the global minimum guaranteed by the previous lemmas must be a root of the polynomial (step 7).

To our knowledge, this is the first proof of the Fundamental Theorem of Algebra in the Boyer–Moore family of theorem provers, but it has been proved earlier in other theorem provers, e.g., Mizar [9], HOL [8], and Coq [7].

2 Continuity and the Extreme Value Theorem

2.1 Continuous Functions

We begin the proof of the FTA with a proof of the Extreme Value Theorem for complex functions. The first step is to define the notion of continuity for complex functions, and this follows the pattern used before for real functions [6]. In particular, we say that f is continuous if for any standard number z, $f(z)$ is close to $f(z_*)$ for any z_* that is close to z. To be precise, two complex numbers are "close" if both the real and imaginary parts of their difference are infinitesimal. This also implies that the distance between the two points is infinitesimal.

We also introduced in ACL2(r) the notion of a continuous function from the complex to the real numbers, and we proved that if both $f : \mathbb{C} \to \mathbb{C}$ and $g : \mathbb{C} \to \mathbb{R}$ are continuous, then so is $g \circ f : \mathbb{C} \to \mathbb{R}$. Moreover, we proved that the function given by the traditional complex norm, $||a+bi|| = \sqrt{a^2+b^2}$, is continuous. So for any continuous function $f : \mathbb{C} \to \mathbb{C}$, the function from $\mathbb{C}$ to $\mathbb{R}$ given by $h(z) = ||f(z)||$ is continuous.

An important difference from the development in [6] is that continuous functions in this paper have two arguments in ACL2, even though we think of them as functions of only one variable. This is similar to the way the function x^n and a^x are introduced in elementary calculus. Those functions are thought of as functions of the single variable x, and the derivatives are given as nx^{n-1} and $a^x \ln a$, even though both functions are simply slices of the bivariate function x^y.

In ACL2(r), we introduce the continuous function `ccfn` using `encapsulate` as `(ccfn context z)`, where the argument `z` is the one that is allowed to vary, whereas `context` is the argument that is held fixed. The non-standard definition of continuity that we use here requires that close points be mapped to close values, but only for standard `z`. Similarly, we require that close points are mapped to close values only when both `z` and `context` are standard.

The `context` argument plays an important role, which can be illustrated by the continuity of polynomials. Consider, for example, a proof that the function x^n is continuous. This would appear to be easy to prove by induction as follows:

1. x^0 is a constant function, so it is definitely continuous.
2. $x^n = x^{n-1} \cdot x$, and x^{n-1} is continuous from the induction hypothesis, and so is the product of continuous functions.

However, this does not work. The formal definition of continuity uses the non-classical notions of *close*: $standard(x) \wedge close(x, x_0) \rightarrow close(x^n, x_0^n)$. But ACL2(r) restricts induction on non-classical functions to standard values of the arguments. In this particular case, we could prove that x^n is continuous, but only for standard values of n, as was done in [10].

Setting that aside for now, we can imagine what would happen if we could establish (somehow) that complex polynomials are continuous. The next step in the proof would be to show that since $p(z)$ is continuous in the complex plane, then $||p(z)||$ is continuous from $\mathbb{C}$ to $\mathbb{R}$, and this could be done by functionally instantiating the previous result about norms of continuous functions. Once more, however, we run into a major complication. The problem is that we want to say this about *all* polynomials, not just about a specific polynomial p. For example, we could use this approach to show that the norm of the following polynomial is continuous:

```
(defun p(z)
 (+ 1 (* #c(0 1) z) (* 3 z z)))
```

But that would lead to a proof that the polynomial p has a root, not that all polynomials have roots. To reason about all polynomials, we use a data structure that can represent any polynomial and an evaluator function that can compute the value of a given polynomial at a given point. For example, the polynomial above is represented with the list `(1 #c(0 1) 3)`, and the function `eval-polynomial` is defined so that `(eval-polynomial '(1 #c(0 1) 3) z) = (p z)`. But we cannot use functional instantiation to show that the norm of `eval-polynomial` is continuous, because the formal statement of continuity uses the non-classical notion of *close*. ACL2(r) restricts functional instantiation so that pseudo-lambda expressions cannot be used in place of functions when the theorem to be proved uses non-classical functions.

It should be noted that both of these restrictions are necessary! For example, x^n is "obviously" continuous, but what happens when n is large? Suppose that $x = 1 + \varepsilon$, where ε is small, so that 1 and $1 + \varepsilon$ are close. From the binomial theorem, $(1+\varepsilon)^n = 1 + n\varepsilon + \cdots$. The thing is that if n is large, $n\varepsilon$ is not necessarily small. For instance, if $n = 1/\varepsilon$, then $n\varepsilon = 1$, and $(1+\varepsilon)^n > 2$, so it is not close to $1^n = 1$. Yes, x^n is continuous, but the non-standard definition of continuity only applies when *x and n* are standard.

Similarly, the restriction for pseudo-lambda expressions is crucial. Consider, for example, the theorem that $f(x)$ is standard when x is standard. This is true, but we could not use it to show that $x + y$ is standard when x is standard by functionally instantiating $\lambda x.x + y$ for f. The proposed theorem is just not true when, for example, $x = 0$ and $y = \varepsilon$.

So these restrictions are important for soundness, and we have to find a way to live with them. In both cases, the key to doing so is the `context` argument. Consider `eval-polynomial`, which has two arguments: the polynomial `poly` and the point `z`. Here, `poly` plays the role of `context`. Notice how `poly` is "held fixed" when we say that a polynomial is continuous over the complex numbers. Because the `context` argument is used to refer to an arbitrary polynomial, there is no need to use a pseudo-lambda expression to introduce `poly`, so we can instantiate the theorem that $||\mathit{eval\text{-}polynomial}(\mathit{poly},z)||$ is continuous from $\mathbb{C}$ to $\mathbb{R}$ without running afoul of the restriction on free variables in functional instantiation of non-classical formulas.

That leaves the continuity of polynomial functions to deal with. Here we want to show that when z is standard and close to z_0, $\mathit{eval\text{-}polynomial}(\mathit{poly},z)$ is also close to $\mathit{eval\text{-}polynomial}(\mathit{poly},z_0)$. Again, the `context` argument plays a major role, because the proof obligation of the `encapsulate` is actually weaker. It says that $\mathit{eval\text{-}polynomial}(\mathit{poly},z)$ must be close to $\mathit{eval\text{-}polynomial}(\mathit{poly},z_0)$ only when both *poly* and z are standard—and in that case we *can* use induction to prove the result. But notice that *poly* is a list, so it is standard when it consists of standard elements and has standard length, which is not the case for all polynomials. As was the case before, this is the best we could expect. Consider, for instance, the polynomial $p(z) = Nz$ where N is a large constant. Then $p(0) = 0$ but $p(\varepsilon) = N\varepsilon$ is not necessarily close to 0.

But what about the Fundamental Theorem of Algebra? Does this mean that we can only prove this theorem for standard arguments? Thankfully, that is not the case. We can proceed with the proof of the Fundamental Theorem of Algebra for *all* polynomials because of the principle of transfer. This works because the statement of the Fundamental Theorem of Algebra is classical: if p is a non-constant, complex polynomial with complex coefficients, then there is some complex number z such that $p(z) = 0$. Notice that statement does not mention concepts like *standard*, *close*, or *small*. So if it holds for all standard polynomials *poly*, it also holds for all polynomials, even the non-standard ones. The same holds for other classical properties, such as the Extreme Value Theorem.

We believe that this approach is significantly cleaner than what we have done before. For example, in [10] we used something similar to the context argument, but the extra arguments were numbers. This means we were reasoning about functions of, say, 10 variables while holding 9 fixed, and if another fixed parameter was needed, we would have to redefine the constrained continuous function. In contrast, the `context` parameter is only required to be standard, not necessarily a number. So we could use it, for example, to hold a list of 9 fixed dimensions in one setting, and 10 fixed dimensions in another one. The constrained function does not need to change to accommodate the extra dimension, and nor do any of the generic theorems that were previously proved, e.g., the Extreme Value Theorem. As long as all the other dimensions are fixed, they can be added to the `context`.

A more traditional way of dealing with this problem is to realize that continuity itself is a classical notion, so we can use a classical constrained function to introduce the notion in ACL2(r). This is what we did in [2] where we explored equivalent definitions of continuity and showed how versions of theorems like the Intermediate Value Theorem could be proved for the classical and non-classical definitions of continuity. But we also think that the approach used here works better, because there is no need to have both classical and non-classical versions of all the theorems. Rather, the necessary theorems can be proved using the non-classical definitions, which are simpler to use in a theorem prover known for rewriting and induction, and then instantiating these (classical) theorems when needed. That is exactly what we do here, where we prove the Extreme Value Theorem using the intuitive notions of *close*, and then instantiate this theorem for all polynomials using the fixed `context` parameter—without having to establish that polynomials satisfy the ε–δ definition of continuity.

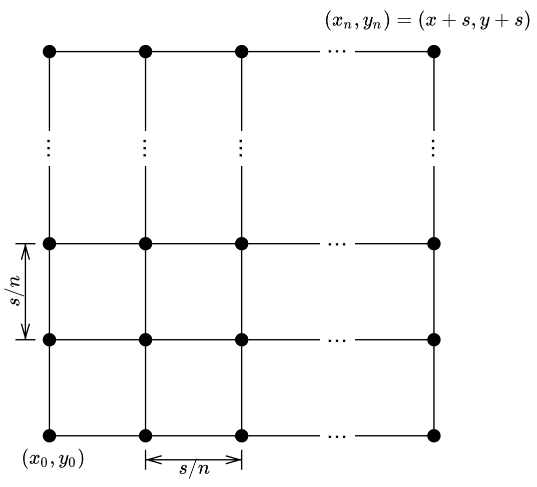


Figure 2: Proof of Extreme Value Theorem

2.2 Proof of the Extreme Value Theorem

We now turn attention to the proof of the Extreme Value Theorem for continuous functions from the complex to the real numbers. The proof is similar in principle to the analogous theorem for real functions, although it is more complicated since the relevant region is a square instead of an interval. In both proofs, we use a constrained function to stand for all possible continuous functions. In the earlier proof, we used `rcfn` to stand for a **r**eal **c**ontinuous **fun**ction. We also use the name `ccfn` to stand for an arbitrary **c**omplex **c**ontinuous **fun**ction. In this section, we will be using `crvcfn` which is a constrained **c**omplex, **r**eal-**v**alued, **c**ontinuous **fun**ction.

Figure 2 shows the basic idea of the proof. The square of size $s \times s$ is subdivided into regions and the value of the function is found at the points in an $(n+1) \times (n+1)$ grid. Using recursion, it is straightforward to find the point on the grid where the function f has a minimal value, say $z_m = (x_i, y_j)$.

At this point, we can assume that (x_0, y_0) and s are both standard, so the square defined by the points (x_0, y_0) and (x_n, y_n) is also standard. This means that if z is inside the square, so is the standard part of z. In particular, the standard part of any of the points in the grid must be inside the square region. And from this point on, we can also assume that n is a large number, so that adjacent points on the grid are a distance at most $\sqrt{2}s/n$ apart, hence close to each other.

Since all the grid points are inside a standard square, they must be limited. This justifies using the non-standard definitional principle `defun-std` to define the minimum as the standard part of z_m, written as ${}^*z_m$ or $standardpart(z_m)$. The non-standard definitional principle assures us that the minimum is precisely ${}^*z_m$, but only when x_0, y_0, and s are standard.

Now, consider any standard point z that lies inside the square region. It must be inside one of the grid cells, and since adjacent grid points are close to each other, z must also be close to the grid points in the enclosing cell, and since it is standard it must be the standard part of those grid points—points that are close to each other have the same standard part. So suppose that z is close to (x_k, y_l)—as just observed, it must be close to *some* point on the grid. There are two cases to consider. If this point happens to be equal to $z_m = (x_i, y_j)$, the minimum point on the grid, then as just observed, it must be the case that its standard part ${}^*z_m$, must be equal to z. So it follows trivially that $f({}^*z_m) \le f(z)$. Conversely, if $(x_k, y_l) \neq (x_i, y_j) =$

z_m, then it must be the case that $f(z_m) = f(x_i, y_j) \leq f(x_k, y_l)$, because z_m was chosen as the minimum point on the grid. Taking standard parts of both sides shows that ${}^*f(z_m) = {}^*f(x_i, y_j) \leq {}^*f(x_k, y_l)$. It is only necessary to prove that ${}^*f(z_0) = f({}^*z_0)$ for all z_0, and again we have established that $f({}^*z_m) \leq f(z)$. So in either case, $f({}^*z_m)$ is at most the value of $f(z)$ for any standard z in the square. Using the transfer principle with `defthm-std`, we generalized this statement to prove that ${}^*z_m$ is the minimum value over all points in the square region from (x_0, y_0) to $(x_0 + s, y_0 + s)$, for all values of x_0, y_0, and s, not just the standard ones.

Although this proof is carried out for arbitrary continuous functions, the intent is to use functional instantiation to show that some specific function achieves its minimum. Thus, it is very helpful to use quantifiers to state the Extreme Value Theorem, so that a future use via functional instantiation does not need to recreate the functions used to search the grid, for example. The following definition captures what it means for the point `zmin` to be the minimum point inside the square or side length `s` with lower-left corner at `z0`:

```
(defun-sk is-minimum-point-in-region (context zmin z0 s)
  (forall (z)
    (implies (and (acl2-numberp z)
                  (acl2-numberp z0)
                  (realp s)
                  (< 0 s)
                  (inside-region-p
                          z
                          (cons (interval (realpart z0)
                                          (+ s (realpart z0)))
                                (interval (imagpart z0)
                                          (+ s (imagpart z0))))))
             (<= (crvcfn context zmin) (crvcfn context z)))))
```

The Extreme Value Theorem, then, simply adds that there is some value of `zmin` that makes this true. The existence is captured with the following definition:

```
(defun-sk achieves-minimum-point-in-region (context z0 s)
  (exists (zmin)
          (implies (and (acl2-numberp z0)
                        (realp s)
                        (< 0 s))
                   (and (inside-region-p
                              zmin
                              (cons (interval (realpart z0)
                                              (+ s (realpart z0)))
                                    (interval (imagpart z0)
                                              (+ s (imagpart z0)))))
                        (is-minimum-point-in-region context
                                                    zmin z0 s)))))
```

Using this definition, the Extreme Value Theorem can be stated simply as follows:

```
(defthm minimum-point-in-region-theorem-sk
  (implies (and (acl2-numberp z0)
                (realp s)
                (< 0 s)
                (inside-region-p z0 (crvcfn-domain))
```

```
            (inside-region-p (+ z0 (complex s s)) (crvcfn-domain)))
      (achieves-minimum-point-in-region context z0 s))
  :hints ...)
```

It is then a simple matter to functionally instantiate this theorem for any polynomial.

3 Growth Lemma for Polynomials

In the previous section, we proved the Extreme Value Theorem for continuous functions from $\mathbb{C}$ to $\mathbb{R}$. In this section, we turn our attention to the values $||p(z)||$ for polynomials p, and we are primarily interested in $||p(z)||$ when z is sufficiently far from the origin.

Let $p(z)$ be given by $p(z) = a_0 + a_1 z + a_2 z^2 + \cdots + a_n z^n$, where $a_n \neq 0$. It follows that

$$||p(z)|| = ||a_0 + a_1 z + a_2 z^2 + \cdots + a_n z^n||.$$

Using induction and the triangle inequality for $||\cdot||$, we proved that

$$||p(z)|| \leq ||a_0|| + ||a_1 z|| + ||a_2 z^2|| + \cdots + ||a_n z^n||.$$

The norm $||\cdot||$ behaves nicely with products. In particular, we proved that $||xy|| = ||x||\,||y||$, so

$$||p(z)|| \leq ||a_0|| + ||a_1||\,||z|| + ||a_2||\,||z^2|| + \cdots + ||a_n||\,||z^n||.$$

Letting $A = \max ||a_i||$, it is easy to prove that

$$||p(z)|| \leq A\left(||z^0|| + ||z^1|| + ||z^2|| + \cdots + ||z^n||\right).$$

We are only interested in the value of $||p(z)||$ when z is sufficiently large, so we can limit the discussion to those values of z with $||z|| \geq 1$. A simple induction shows that for those z, $||z|| \leq ||z^n||$, so

$$||p(z)|| \leq A\left(||z^n|| + ||z^n|| + ||z^n|| + \cdots + ||z^n||\right) = A(n+1)||z^n||.$$

Another simple induction shows that $||z^n|| = ||z||^n$, so we have that

$$||p(z)|| \leq A(n+1)||z||^n.$$

Now, suppose that $\frac{A(n+1)}{K} \leq ||z||$ for an arbitrary, real number K—clearly, for any fixed K, we can always find a z large enough to make this true, since A and n are always fixed. Then $A(n+1) \leq K||z||$, and we have

$$||p(z)|| \leq A(n+1)||z||^n \leq K||z||^{n+1}.$$

The last result holds for all polynomials p. Now we use that result to get both lower and upper bounds for any polynomial q. In particular, let $q(z) = a_0 + a_1 z + a_2 z^2 + \cdots + a_n z^n$. Then q can be written as $q(z) = (a_0 + a_1 z + a_2 z^2 + \cdots + a_{n-1} z^{n-1}) + a_n z^n$. Applying the previous result to the polynomial in parentheses, we have that

$$q(z) \leq K||z||^n + ||a_n||\,||z||^n,$$

for any K and z such that $An \leq K||z||$. Choosing $K = ||a_n||/2$, we have that

$$q(z) \leq K||z||^n + ||a_n||\,||z||^n = \frac{||a_n||}{2}||z||^n + ||a_n||\,||z||^n = \frac{3}{2}||a_n||\,||z||^n.$$

That provides a nice upper bound for $||q(z)||$. In ACL2, this can be shown as follows.

```
(defthm upper-bound-for-norm-poly
  (implies (and (polynomial-p poly)
                (< 1 (len poly))
                (not (equal (leading-coeff poly) 0))
                (acl2-numberp z)
                (<= 1 (norm2 z))
                (<= (/ (* 2
                          (max-norm2-coeffs (all-but-last poly))
                          (1- (len poly)))
                       (norm2 (leading-coeff poly)))
                    (norm2 z)))
           (<= (norm2 (eval-polynomial poly z))
               (* 3/2
                  (norm2 (leading-coeff poly))
                  (expt (norm2 z) (1- (len poly))))))
  :hints ...)
```

We can find a lower bound on $||q(z)||$ by observing that

$$q(z) = a_n z^n - (-a_0 - a_1 z - a_2 z^2 + \cdots - a_{n-1} z^{n-1}),$$

and using the variant of the triangle inequality $||a-b|| \geq ||a| - ||b||$, we have that

$$||q(z)|| \geq ||a_n z^n|| - || - a_0 - a_1 z - a_2 z^2 + \cdots - a_{n-1} z^{n-1}||.$$

Further, since multiplying a value by -1 does not change its norm, we can write this as

$$||q(z)|| \geq ||a_n z^n|| - ||a_0 + a_1 z + a_2 z^2 + \cdots + a_{n-1} z^{n-1}||.$$

As we saw previously, $||a_0 + a_1 z + a_2 z^2 + \cdots + a_{n-1} z^{n-1}|| \leq \frac{||a_n||}{2} ||z||^n$, so

$$||q(z)|| \geq ||a_n z^n|| - ||a_0 + a_1 z + a_2 z^2 + \cdots + a_{n-1} z^{n-1}|| \geq ||a_n z^n|| - \frac{||a_n||}{2} ||z||^n = \frac{1}{2} ||a_n|| \, ||z||^n.$$

This can be shown in ACL2 as follows:

```
(defthm lower-bound-for-norm-poly
  (implies (and (polynomial-p poly)
                (< 1 (len poly))
                (not (equal (leading-coeff poly) 0))
                (acl2-numberp z)
                (<= 1 (norm2 z))
                (<= (/ (* 2
                          (max-norm2-coeffs (all-but-last poly))
                          (1- (len poly)))
                       (norm2 (leading-coeff poly)))
                    (norm2 z)))
           (<= (* 1/2
                  (norm2 (leading-coeff poly))
                  (expt (norm2 z) (1- (len poly))))
               (norm2 (eval-polynomial poly z))))
  :hints ...)
```

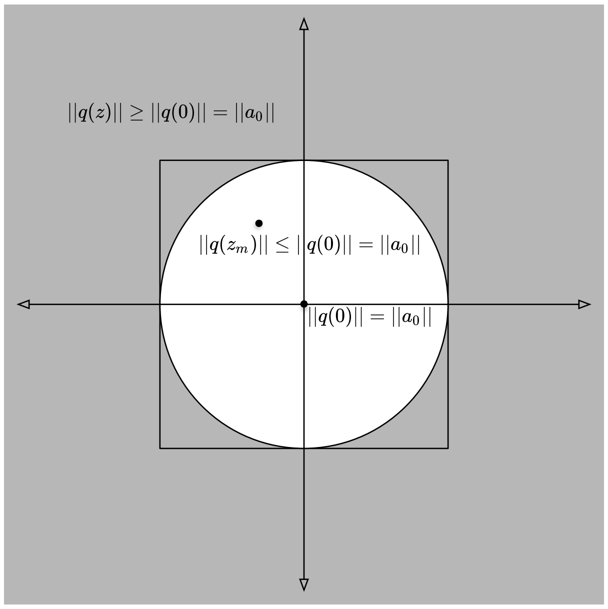

$||q(z)|| \geq ||q(0)|| = ||a_0||$
$||q(z_m)|| \leq ||q(0)|| = ||a_0||$
$||q(0)|| = ||a_0||$

Figure 3: Local Minimum Is a Global Minimum

Combining the previous two results, we have shown that

$$\frac{1}{2}||a_n||\,||z||^n \leq ||q(z)|| \leq \frac{3}{2}||a_n||\,||z||^n,$$

for values of z with sufficiently large $||z||$.

Since the results hold for all z with sufficiently large $||z||$, we can restrict ourselves to z such that $||z|| \geq 2\frac{||a_0||}{||a_n||}$. In this case, we have that

$$q(z) \geq \frac{1}{2}||a_n||\,||z||^n \geq \frac{1}{2}||a_n||\,||z|| \geq \frac{1}{2}||a_n||\left(2\frac{||a_0||}{||a_n||}\right) = ||a_0||.$$

But observe that $q(0) = a_0$, so $||q(z)|| \geq ||q(0)||$.

The situation is summarized in Figure 3. All points in the gray area—that is, outside of the circle—are large enough that $||q(z)|| \geq ||a_0|| = ||q(0)||$. On the other hand, the minimum point z_m guaranteed by the Extreme Value Theorem lies inside the square, and since it is the minimum point inside the square, it holds that $||q(z_m)|| \leq ||q(0)|| = ||a_0||$. Combining these two statements, we have that $||q(z_m)|| \leq ||q(z)||$ for all z whether they are inside or outside of the circle. In other words, z_m is not just a minimum inside the square region. It is a *global* minimum for q.

4 d'Alembert's Lemma

We now prove d'Alembert's Lemma, which states that for a non-constant polynomial p, if z is such that $p(z) \neq 0$, then there is some z_0 such that $||p(z_0)|| < ||p(z)||$. In particular, if $p(z) \neq 0$ then z cannot be a global minimum of $||p(\cdot)||$.

We prove this lemma by proceeding in a series of special cases. First, let's assume that $a_0 = 1$ so that

$$p(z) = 1 + a_1 z + a_2 z^2 + \cdots + a_n z^n.$$

Let k be the largest index such that $a_1 = a_2 = \cdots = a_{k-1} = 0$. We note that $k \leq n$, since we assume throughout this paper that $a_n \neq 0$. Then,

$$\begin{aligned} p(z) &= 1 + a_k z^k + a_{k+1} z^{k+1} + \cdots + a_n z^n \\ &= 1 + a_k z^k + z^{k+1} q(z) \end{aligned}$$

where q is a particular polynomial. Of course, this means that

$$\begin{aligned} ||p(z)|| &= ||1 + a_k z^k + z^{k+1} q(z)|| \\ &\leq ||1 + a_k z^k|| + ||z^{k+1} q(z)|| \end{aligned}$$

Suppose that s is real number such that $0 < s < 1$, and let $z_s = \sqrt[k]{-\frac{s}{a_k}}$. We note in passing that the existence of such a z_s is guaranteed by de Moivre's lemma, which we proved as part of this effort, using the principal logarithmic function defined in [5].

```
(defthm de-moivre-1
  (implies (and (acl2-numberp z)
                (natp n))
           (equal (expt z n)
                  (* (expt (radiuspart z) n)
                     (acl2-exp (* #c(0 1) n (anglepart z))))))
  :hints ...)
(defthm de-moivre-2
  (implies (and (acl2-numberp z)
                (posp n))
           (equal (expt (nth-root z n) n) z))
  :hints ...)
```

Since $z_s = \sqrt[k]{-\frac{s}{a_k}}$, we have that

$$a_k z_s^k = a_k \left(\sqrt[k]{-\frac{s}{a_k}} \right)^k = -s.$$

So $||1 + a_k z_s^k|| = ||1 - s|| = |1 - s| = 1 - s$, since s is real and $0 < s < 1$. This means that

$$\begin{aligned} ||p(z_s)|| &\leq ||1 + a_k z_s^k|| + ||z_s^{k+1} q(z_s)|| \\ &= 1 - s + ||z_s^{k+1} q(z_s)|| \\ &= 1 - s + ||z_s||^k \, ||z_s|| \, ||q(z_s)|| \\ &= 1 - s + \frac{s}{||a_k||} ||z_s|| \, ||q(z_s)|| \\ &= 1 - s \left(1 - \frac{||z_s||}{||a_k||} ||q(z_s)|| \right) \end{aligned}$$

It's obvious that the term in parentheses is real and at most 1. What we want to show now is that it is actually positive and at most 1 for at least one choice of s, and hence z_s. That is, we will show that there is an s such that $\frac{||z_s||}{||a_k||} ||q(z_s)|| < 1$.

We find this s by observing that whenever z is such that $0 < ||z|| < 1$, then $||q(z)|| \leq M(n+1)$, where M is the maximum norm of the coefficients, i.e., $M = \max ||a_i||$. Let $r = \frac{||z||}{||a_k||}||q(z)||$. If $q(z) = 0$, then $r = 0 < 1$. Otherwise, as long as $0 < ||z|| < 1$ and $0 < ||z|| < \frac{||a_k||}{M(n+1)}$, we have that $r = \frac{||z||}{||a_k||}||q(z)|| < \frac{||a_k||}{||a_k||M(n+1)}||q(z)|| \leq \frac{1}{M(n+1)}M(n+1) = 1$. So as long as $0 < ||z|| < 1$ and $0 < ||z|| < \frac{||a_k||}{M(n+1)}$, $0 \leq r < 1$ (regardless of the value of $q(z)$.) Therefore,

$$\begin{aligned}
||p(z)|| &\leq 1 - s\left(1 - \frac{||z||}{||a_k||}||q(z)||\right) \\
&\leq 1 - s(1-r) \\
&< 1.
\end{aligned}$$

The last inequality follows since $0 \leq r < 1$, so $1 - r > 0$ and $s > 0$. So it remains only to choose an s such that $0 < ||z_s|| < \frac{||a_k||}{M(n+1)}$. Since $z_s = \sqrt[k]{-\frac{s}{a_k}}$ this is equivalent to finding a positive number s small enough such that $s < \frac{||a_k||^{k+1}}{M^k(n+1)^k}$, which is obviously possible since the right-hand side is positive. This is summarized in the ACL2 theorem below:

```
(defthm lowest-exponent-split-10
  (implies (and (polynomial-p poly)
                (equal (car poly) 1)
                (< 1 (len poly))
                (not (equal (leading-coeff poly) 0))
                (equal (car poly) 1)
                )
           (< (norm2 (eval-polynomial
                       poly
                       (fta-bound-1 poly
                                    (input-with-smaller-value
                                         poly))))
              1))
  :hints ...)
```

The function `fta-bound-1` corresponds to the choice of x_s, and `input-with-smaller-value` finds a suitable value of s.

To complete the proof, observe that since $||p(z_s)|| < 1 = p(0)$, we have that 0 cannot be the global minimum. It is worth remembering that the only thing we assumed about this polynomial is that $a_0 = p(0) = 1$. We generalize this proof by removing this assumption and letting $a_0 \neq 1$. If it happens that $a_0 = 0$, then $p(0) = a_0 = 0$ and the polynomial has a root. Otherwise, $a_0 \neq 0$ and we can define the new polynomial

$$p_1(z) = 1 + \frac{a_1}{a_0}z + \frac{a_2}{a_0}z^2 + \cdots + \frac{a_n}{a_0}z^n.$$

Notice that $p_1(z) = p(z)/a_0$, so $||p_1(z)|| = ||p(z)||/||a_0||$. This means that if 0 is not a global minimum of $||p||$, it cannot be a global minimum of $||p_1||$, either.

Finally, we remove the assumption that it is at the point $z = 0$ that $p(z) \neq 0$. Suppose, in fact, that $p(z_0) \neq 0$, and define

$$\begin{aligned}
p_2(z) &= p(z+z_0) \\
&= a_0 + a_1(z+z_0) + a_2(z+z_0)^2 + \cdots + a_n(z+z_0)^n \\
&= b_0 + b_1 z + b_2 z^2 + \cdots + b_n z^n
\end{aligned}$$

This is a more complicated transformation than the one defining p_1, but we showed that p_2 is in fact a polynomial, that its highest exponent is n, and that if $a_n \neq 0$ then $b_n \neq 0$ (in fact, $b_n = a_n$). That means that $p_2(0) = p(z_0) \neq 0$, and we can apply the previous theorem to show that z_0 is not a global minimum of $||p_2||$.

There is one remaining assumption. Throughout this paper, we have been assuming that the leading coefficient $a_n \neq 0$. But what about a polynomial such as

$$p(z) = a_0 + a_1 z + a_2 z^2 + \cdots + a_{n-1} z^{n-1} + 0 z^n$$

To generalize the results to this polynomial, we construct a new polynomial $trunc(p)$ that simply removes all terms a_i where $a_i = a_{i+1} = \cdots = a_n = 0$. It is easy to show that $trunc(p)$ is a polynomial and that it agrees with p at all values of z. Moreover, we say that a polynomial p is not constant if at least one of $a_1, a_2, \ldots, a_n$ is not equal to 0. Then it is easy to prove that $trunc(p)$ is also a non-constant polynomial, such that its leading coefficient is not zero. That means that the theorems above apply to $trunc(p)$, and thus to p.

This completes the proof of d'Alembert's Lemma: if p is a polynomial and z is such that $p(z) \neq 0$, then z is not a global minimum of p. It is then trivial to prove the Fundamental Theorem of Algebra. Since we already know that there is a z_m such that $||p(\cdot)||$ has a minimum at z_m, it must be the case that $p(z_m) = 0$.

In ACL2, we can write the condition that a given polynomial has a root using the following Skolem definition:

```
(defun-sk polynomial-has-a-root (poly)
  (exists (z)
          (equal (eval-polynomial poly z) 0)))
```

The Fundamental Theorem of Algebra can then be stated as follows:

```
(defthm fundamental-theorem-of-algebra-sk
  (implies (and (polynomial-p poly)
                (not (constant-polynomial-p poly)))
           (polynomial-has-a-root poly))
  :hints ...)
```

5 Conclusion

We have shown a proof in ACL2(r) of the Fundamental Theorem of Algebra. Table 1 gives an indication of the size of the proof in ACL2(r). The proof follows the 1746 proof of d'Alembert, which was maligned by Gauss on the grounds that it was not sufficiently rigorous. Gauss was right, in that it took another hundred years for the notion of continuity to be sufficiently rigorous to prove the Extreme Value Theorem, on which the proof depends. Gauss offered a solution to d'Alembert's dilemma, in the form of a more formal proof of the Fundamental Theorem of Algebra, which nevertheless suffered from its own lack of rigor with respect to algebraic curves. Aware of this, Gauss proceeded to offer three other proofs of the Fundamental Theorem of Algebra, all essentially correct. We are in the midst of studying complex polynomials in ACL2, and Gauss's proofs offer a fertile playground for this purpose, so we expect to formalize some of those proofs in ACL2(r) in the future.

File	**Description**	**#Definitions**	**#Theorems**
`norm2`	Basic facts about the complex norm	2	72
`complex-lemmas`	Basic facts from complex analysis	0	25
`de-moivre`	deMoivre's theorem	2	36
`complex-continuity`	Extreme value theorem for complex functions	19	96
`complex-polynomials`	Polynomials are continuous, achieve a minimum in a closed area, and have arbitrarily large values for large enough arguments; d'Alembert's Lemma; and Fundamental Theorem of Algebra	41	197
Total		64	426

Table 1: Proof Statistics

References

[1] Harel Cain (2018): *C. F. Gauss's Proofs of the Fundamental Theorem of Algebra*. Available at `http://math.huji.ac.il/~ehud/MH/Gauss-HarelCain.pdf`.

[2] J. Cowles & R. Gamboa (2014): *Equivalence of the Traditional and Non-Standard Definitions of Concepts from Real Analysis*. In: *Proceedings of the 12th International Workshop of the ACL2 Theorem Prover and its Applications*, doi:10.1007/3-540-36126-X_17.

[3] B. Fine & G. Rosenberger (1997): *The Fundamental Theorem of Algebra*. Undergraduate Texts in Mathematics, Springer New York, doi:10.1007/978-1-4612-1928-6.

[4] Fundamental theorem of algebra (2001): *Fundamental theorem of algebra — Wikipedia, The Free Encyclopedia*. Available at `https://en.wikipedia.org/wiki/Fundamental_theorem_of_algebra`.

[5] R. Gamboa & J. Cowles (2009): *Inverse Functions in ACL2(r)*. In: *Proceedings of the Eighth International Workshop of the ACL2 Theorem Prover and its Applications (ACL2-2009)*, doi:10.1145/1637837.1637846.

[6] R. Gamboa & M. Kaufmann (2001): *Nonstandard analysis in ACL2*. *Journal of Automated Reasoning* 27(4), pp. 323–351, doi:10.1023/A:1011908113514.

[7] H. Geuvers, F. Wiedijk, J. Zwanenburg, R. Pollack & Ha Barendregt: *The "Fundamental Theorem of Algebra" Project*. Available at `http://www.cs.kun.nl/~freek/fta/index.html`.

[8] J. Harrison (2001): *Complex Quantifier Elimination in HOL*. In: *TPHOLs 2001: Supplemental Proceedings*, pp. 159–174. Available at `http://www.inf.ed.ac.uk/publications/online/0046/b159.pdf`.

[9] R. Milewski (2000): *Fundamental Theorem of Algebra*. In: *Journal of Formalized Mathematics*, 12.

[10] J. Sawada & R. Gamboa (2002): *Mechanical Verification of a Square Root Algorithm using Taylor's Theorem*. In: *Formal Methods in Computer-Aided Design (FMCAD'02)*, doi:10.1007/3-540-36126-X_17.

[11] Daniel J. Velleman (2015): *The Fundamental Theorem of Algebra: A Visual Approach*. *The Mathematical Intelligencer* 37(4), pp. 12–21, doi:10.1007/s00283-015-9572-7.

Real Vector Spaces and the Cauchy-Schwarz Inequality in ACL2(r)

Carl Kwan Mark R. Greenstreet

Department of Computer Science
University of British Columbia[*]
Vancouver, Canada
{carlkwan,mrg}@cs.ubc.ca

We present a mechanical proof of the Cauchy-Schwarz inequality in ACL2(r) and a formalisation of the necessary mathematics to undertake such a proof. This includes the formalisation of $\mathbb{R}^n$ as an inner product space. We also provide an application of Cauchy-Schwarz by formalising $\mathbb{R}^n$ as a metric space and exhibiting continuity for some simple functions $\mathbb{R}^n \to \mathbb{R}$.

The Cauchy-Schwarz inequality relates the magnitude of a vector to its projection (or inner product) with another:

$$|\langle u,v\rangle| \le \|u\|\|v\|$$

with equality iff the vectors are linearly dependent. It finds frequent use in many branches of mathematics including linear algebra, real analysis, functional analysis, probability, etc. Indeed, the inequality is considered to be among "The Hundred Greatest Theorems" and is listed in the "Formalizing 100 Theorems" project. To the best of our knowledge, our formalisation is the first published proof using ACL2(r) or any other first-order theorem prover.

1 Introduction

The Cauchy-Schwarz inequality is considered to be one of the most important inequalities in mathematics. Indeed, it appears in functional analysis, real analysis, probability theory, linear algebra, and combinatorics to name a few. Cauchy-Schwarz even made an appearance on an online list of "The Hundred Greatest Theorems" and the subsequent formalised version of the list "Formalizing 100 Theorems" [8, 24]. Some of the systems used for the proof include the usual suspects HOL/Isabelle, Coq, Mizar, PVS, etc. Notably missing, however, from the list of formalisations of Cauchy-Schwarz is a proof in ACL2 or ACL2(r). We remedy this.

In this paper, we present a formal proof of the Cauchy-Schwarz inequality in ACL2(r) including both forms (squared and norm versions) and the conditions for equality. This is the first proof of Cauchy-Schwarz for real vector spaces of arbitrary dimension $n \in \mathbb{N}$ in ACL2(r); in fact, to the best of our knowledge, this is the first proof in any first-order theorem prover. Such a formalisation suggests ACL2(r) applications in the various areas of mathematics in which the inequality appears. Indeed, we use Cauchy-Schwarz to prove $\mathbb{R}^n$ is a metric space in this paper and to prove theorems involving convex functions in [10].

The proof of Cauchy-Schwarz requires a theory of real vectors in ACL2(r). ACL2(r) extends ACL2 with real numbers formalised via non-standard analysis [3], and it supports automated reasoning involving irrational real and complex numbers in addition to the respective rational subsets that are supported

[*]This work is supported in part by the National Science and Engineering Research Council of Canada (NSERC) Discovery Grant program and the Institute for Computing, Information and Cognitive Systems (ICICS) at UBC.

Shilpi Goel and Matt Kaufmann (Eds.):
ACL2 Theorem Prover and its Applications (ACL2-2018)
EPTCS 280, 2018, pp. 111–127, doi:10.4204/EPTCS.280.9

in the vanilla distribution of ACL2 [5]. The natural next step beyond $\mathbb{R}$ is $\mathbb{R}^n$ which is fundamental to many branches of mathematics. Indeed, as a geometric structure, we view $\mathbb{R}^n$ as a place in which to perform analysis; as an algebraic object, $\mathbb{R}^n$ is a direct sum of subspaces. Under different lenses we view $\mathbb{R}^n$ as different structures, and thus it inherits the properties of the related structures. It is a metric space, a Hilbert space, a topological space, a group, etc. The ubiquitous nature of $\mathbb{R}^n$ suggests such a formalisation will open opportunities for applications in the many areas of mathematics $\mathbb{R}^n$ appears.

Formalising $\mathbb{R}^n$ for arbitrary n introduces technical difficulties in ACL2(r). The fundamental differences between the rational and irrational numbers induce a subtle schism between ACL2 and ACL2(r) wherein the notions formalised in ACL2 (which are bestowed the title of *classical*) are far more well-behaved than those unique to ACL2(r) (which are respectively referred to as *non-classical*). The arbitrariness of n suggests the necessity of defining operations recursively – yet non-classical recursive functions are not permitted in ACL2(r).

For the purposes of this paper, we present $\mathbb{R}^n$ formalised from two perspectives. First, we consider $\mathbb{R}^n$ as an inner product space under the usual dot product. The second perspective formalises $\mathbb{R}^n$ as a metric space. In this case, we use the usual Euclidean metric. Metrics also provide the framework in which to perform analysis and we introduce notions of continuity in ACL2(r) for functions on $\mathbb{R}^n$ and provide some simple examples of such functions.

Verifying the metric space properties of $\mathbb{R}^n$ traditionally requires the Cauchy-Schwarz inequality, which is the highlight of this paper. Accordingly, a focus on the formalisation of Cauchy-Schwarz will be emphasised. For the sake of brevity, a selection of the remaining definitions and theorems will be showcased with most details omitted. In the remainder of the paper we instead discuss the dynamics between approaching formalisation via simple definitions and avoiding fundamental logical limitations in expressibility. For example, real vectors are unsurprisingly represented using lists of ACL2(r) real numbers. However, we were surprised to find that deciding whether all components of a vector were infinitesimals via a recursive function that applies a recognizer to each entry was impossible due to the prohibition of non-standard recursive functions. We provide our solutions to this issue and other challenges that arose in the course of the formalisation.

Most of the mathematics encountered in this paper can be found in standard texts such as [20, 21, 11, 7]. Our proof Cauchy-Schwarz is similar to the one in [19, Chapter 9] and further reading specific to relevant inequalities can be found in [22]. Non-standard analysis is less standard of a topic but some common references are [18, 3, 12].

2 Related Work

Theorem provers with prior formalisations of Cauchy-Schwarz include HOL Light, Isabelle, Coq, Mizar, Metamath, ProofPower, and PVS. However, it appears some of the statements do not mention the conditions for equality [13, 9].

While this paper focuses on $\mathbb{R}^n$ from two perspectives, most other formalisations in the literature outline only one view of $\mathbb{R}^n$ – usually either neglecting to address metrics or lacking in the development of vectors. Moreover, to the best of our knowledge, these formalisations of $\mathbb{R}^n$ are verified in a higher-order setting. For, example, a theory of complex vectors with a nod towards applications in physics was formalised in HOL Light but does not address metrics [2]. On the other end of the spectrum, there is a HOL formalisation of Euclidean metric spaces and abstract metric spaces in general that does not fully include a theory of real vectors [6, 14]. This observation extends to similar results in Coq [23].

Within ACL2(r), there has been formalisation for some special cases of n. The work that handled

$n = 1$ is indeed fundamental and indispensable [5]. We may view $\mathbb{C} \simeq \mathbb{R}^2$ as a vector space over $\mathbb{R}$ so $n = 2$ is immediate since ACL2(r) supports complex numbers. Moreover, extensions of $\mathbb{C}$ such as the quaternions $\mathbb{H}$ and the octonions $\mathbb{O}$ with far richer mathematical structure than typical vector spaces have recently been formalised, which addresses the cases of $n = 4$ and $n = 8$ [4].

3 Preliminaries

3.1 Inner Product Spaces

Inner product spaces are vector spaces equipped with an inner product which induces – among other notions – rigorous definitions of the *angle* between two vectors and the *size* of a vector. More formally, a vector space is a couple (V, F) where V is a set of vectors and F a field equipped with scalar multiplication such that

$$
\begin{aligned}
&v + (u + w) = (v + u) + w && \textit{(associativity)} && (1)\\
&v + u = u + v && \textit{(commutativity)} && (2)\\
&\text{there exists a } 0 \in V \text{ such that } v + 0 = v && \textit{(additive identity)} && (3)\\
&\text{there exists a } -v \in V \text{ such that } v + (-v) = 0 && \textit{(additive inverse)} && (4)\\
&a(bv) = (ab)v && \textit{(compatibility)} && (5)\\
&1v = v && \textit{(scalar identity)} && (6)\\
&a(v + u) = av + au && \textit{(distributivity of vector addition)} && (7)\\
&(a + b)v = av + bv && \textit{(distributivity of field addition)} && (8)
\end{aligned}
$$

for any $v, u, w \in V$ and any $a, b \in F$ [19, Chapter 1].

An inner product space is a triple $(V, F, \langle -, - \rangle)$ where (V, F) is a vector space and $\langle -, - \rangle : V \times V \to F$ is a function called an inner product, i.e. a function satisfying

$$
\begin{aligned}
&\langle au + v, w \rangle = a\langle u, w \rangle + \langle v, w \rangle && \textit{(linearity in the first coordinate)} && (9)\\
&\langle u, v \rangle = \langle v, u \rangle \text{ when } F = \mathbb{R} && \textit{(commutativity)} && (10)\\
&\langle u, u \rangle \geq 0 \text{ with equality iff } u = 0 && \textit{(positive definiteness)} && (11)
\end{aligned}
$$

for any $u, v, w \in V$ and $a \in F$ [19, Chapter 9].

For $\mathbb{R}^n$, this means $(\mathbb{R}^n, \mathbb{R}, \langle -, - \rangle)$ is our inner product space and we choose $\langle -, - \rangle$ to be the bilinear map

$$\langle (u_1, u_2, \ldots, u_n), (v_1, v_2, \ldots, v_n) \rangle = u_1 v_1 + u_2 v_2 + \cdots u_n v_n \tag{12}$$

also known as the dot product.

3.2 The Cauchy-Schwarz Inequality

Let $\|\cdot\|$ be the norm induced by $\langle -, - \rangle$. The Cauchy-Schwarz inequality states

$$|\langle u, v \rangle|^2 \leq \langle u, u \rangle \cdot \langle v, v \rangle \tag{Cauchy-Schwarz I}$$

or, equivalently,

$$|\langle u, v \rangle| \leq \|u\| \|v\| \tag{Cauchy-Schwarz II}$$

for any vectors u, v [19, Chapter 9]. Moreover, equality holds iff u, v are linearly dependent.

Proof (sketch). If $v = 0$, then the claims are immediate. Suppose $v \neq 0$ and let a be a field element. Observe

$$0 \leq \|u-av\|^2 = \langle u-av, u-av\rangle = \|u\|^2 - 2a\langle u,v\rangle + a^2\|v\|^2. \tag{13}$$

Setting $a = \|v\|^{-2}\langle u,v\rangle$ and rearranging produces (Cauchy-Schwarz I). Take square roots and we get (Cauchy-Schwarz II). Note that $0 \leq \|u-av\|^2$ is the only step with an inequality so there is equality iff $u = av$. □

More details will be provided as we discuss the formal proof – especially the steps that involve "rearranging".

3.3 Metric Spaces

Metric spaces are topological spaces equipped with a metric function which is a rigorous approach to defining the intuitive notion of *distance* between two vectors. Formally, a metric space is a couple (M,d) where M is a set and $d : M \times M \to \mathbb{R}$ a function satisfying

$$d(x,y) = d(y,x) \qquad \textit{(commutativity)} \tag{14}$$

$$d(x,y) \geq 0 \text{ with equality iff } x = y \qquad \textit{(positive definiteness)} \tag{15}$$

$$d(x,y) \leq d(x,z) + d(z,y) \qquad \textit{(triangle inequality)} \tag{16}$$

for any $x,y,z \in M$ [19, Chapter 9]. The function d is called a *metric*.

In this case, we view $\mathbb{R}^n$ as $(\mathbb{R}^n, d_2)$ where $d_2 : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ denotes the Euclidean metric

$$d_2(x,y) = \sqrt{(x_1-y_1)^2 + (x_2-y_2)^2 + \cdots + (x_n-y_n)^2} = \left(\sum_{i=1}^{n}(x_i-y_i)^2\right)^{1/2}. \tag{17}$$

Proofs for the first two properties of the metric d_2 follow directly from the definition of the function. The triangle inequality follows from the Cauchy-Schwarz inequality [11, Chapter 15].

A metric provides sufficient tools for defining continuity in a manner similar to that in single variable calculus [20, Chapter 4]. A function $f : \mathbb{R}^n \to \mathbb{R}$ is *continuous* everywhere if for any $x \in \mathbb{R}^n$ and $\varepsilon > 0$ there is a $\delta > 0$ such that for any $y \in \mathbb{R}^n$, if

$$d_2(x,y) < \delta, \tag{18}$$

then

$$|f(x) - f(y)| < \varepsilon. \tag{19}$$

This naturally leads to differentiability which is mentioned briefly in [10].

3.4 Non-standard Analysis and ACL2(r)

Classical real analysis is well known for its epsilon-delta approach to mathematical theory-building. For example, we say that function $f : \mathbb{R} \to \mathbb{R}$ is continuous at $x \in \mathbb{R}$ iff [20, Chapter 4]

$$\forall\varepsilon > 0,\ \exists\delta > 0 :\ \forall y \in \mathbb{R},\ |y-x| < \delta \implies |f(y) - f(x)| < \varepsilon. \tag{20}$$

This classical approach makes extensive use of nested quantifiers and support for quantifiers in ACL2 and ACL2(r) is limited. In fact, proofs involving terms with quantifiers often involve recursive witness functions that enumerate all possible values for the quantified term, e.g. see [1]. Of course, we cannot enumerate all of the real numbers. Instead of using epsilon-delta style reasoning, ACL2(r) is built on a formalisation of non-standard analysis – a more algebraic yet isomorphic approach to the theory of real analysis [3].

Non-standard analysis introduces an extension of $\mathbb{R}$ called the *hyperreals* $^*\mathbb{R} \supset \mathbb{R}$ which include numbers larger in magnitude than any finite real and the reciprocals of such numbers. These large hyperreals are aptly named *infinite* and their reciprocals are named *infinitesimal*. If ω is an infinite hyperreal, then it follows that $|1/\omega| < x$ for any positive finite real x. Also, 0 is an infinitesimal.

Any finite hyperreal is the sum of a real number and an infinitesimal. The real part of a hyperreal can be obtained through the *standard-part* function $\mathrm{st} : {}^*\mathbb{R} \to \mathbb{R}$.

To state a function $f : \mathbb{R}^n \to \mathbb{R}$ is continuous in the language of non-standard analysis amounts to: if $d(x,y)$ is an infinitesimal for a standard x, then so is $|f(x) - f(y)|$.

In ACL2(r), lists of real numbers are recognized by `real-listp`. The recognizer for infinitesimals is the function `i-small` and the recognizer for infinite hyperreals is `i-large`. Reals that are not `i-large` are called `i-limited`.

4 $\mathbb{R}^n$ as an Inner Product Space in ACL2(r)

4.1 Vector Space Axioms

Most of the properties of real vector spaces pass with relative ease and minimal guidance by the user. Vector addition is `(vec-+ u v)` and scalar multiplication is `(scalar-* a v)` where a is a field element and u, v are vectors in $\mathbb{R}^n$. The zero vector is recognized by `zvecp`.

One slightly more challenging set of theorems involve vector subtraction, `(vec-- u v)`. Ideally, vector subtraction would be defined as a macro equivalent to `(vec-+ u (scalar-* -1 v))` to remain consistent with subtraction for reals in ACL2(r) and we indeed do so. However, it turns out that proving theorems regarding a function equivalent to `vec--`, which we call `vec--x`, and then proving the theorems for `vec--` via the equivalence is more amenable to verification in ACL2(r) than immediately proving theorems about `vec--`. In particular, the theorems involving closure, identity, inverses, anticommutativity, etc. are almost immediate using this approach. An example of this can be seen in Program 4.1. Upon verification of the desired properties for `vec--`, the theorem positing the equivalence of `vec--` to `vec--x` is disabled so as to not pollute the space of rules.

4.2 Inner Product Space Axioms

Like the vector space axioms, the majority of the relevant inner product space theorems passes by guiding ACL2(r) through textbook proofs. Aside from the usual suspects, one notable set of theorems are the bilinearity of the dot product. In particular, while the proof for the linearity of the first coordinate of the dot product executes via induction without any hints, the proof of linearity for the second coordinate does not pass so easily. Providing an induction scheme in the form of a hint would likely produce the desired proof. It was simpler, however, to apply commutativity of the dot product and use linearity of the first coordinate to exhibit the same result, ie. given

$$\langle au, v\rangle = a\langle u, v\rangle, \tag{21}$$

$$\langle u, v\rangle = \langle v, u\rangle, \tag{22}$$

Program 4.1 Using the equivalence between `vector--` and `vec---x`.

```
(defthm vec---equivalence
 (implies (and (real-listp vec1)
               (real-listp vec2)
               (= (len vec1) (len vec2)))
          (equal (vec-- vec1 vec2) (vec--x vec1 vec2)))
 :hints (("GOAL" :in-theory (enable vec--x vec-+ scalar-*)
                 :induct (and (nth i vec1) (nth i vec2)))))
...
(defthm vec--x-anticommutativity
        (= (vec--x vec1 vec2) (scalar-* -1 (vec--x vec2 vec1)))
 :hints (("GOAL" :in-theory (enable vec--x scalar-*))))

(defthm vec---anticommutativity
 (implies (and (real-listp vec1) (real-listp vec2)
               (= (len vec2) (len vec1)))
          (= (vec-- vec1 vec2) (scalar-* -1 (vec-- vec2 vec1))))
 :hints (("GOAL" :use ((:instance vec---equivalence)
                       (:instance vec---equivalence (vec1 vec2) (vec2 vec1))
                       (:instance vec--x-anticommutativity)))))
```

we have

$$\langle u, av\rangle = \langle av, u\rangle = a\langle v, u\rangle = a\langle u, v\rangle. \quad (23)$$

Program 4.2 shows the ACL2(r) version of this proof.

Our reliance of proving theorems about algebraic structures via their algebraic properties instead of via induction is well exemplified here. This is especially important for the following formalisation of metric spaces. Because non-classical recursive functions are not permitted in ACL2(r), suppressing the definition of recursive functions on vectors, say `dot`, within a `define` facilitates the reasoning of infinitesimals in the space of $\mathbb{R}^n$. In particular, since $\langle -,-\rangle : \mathbb{R}^n \to \mathbb{R}$ is a real-valued function, we may connect the notion of infinitesimal values of $\langle -,-\rangle$ with the entries of the vectors on which $\langle -,-\rangle$ is evaluated without unravelling the recursive definition of `dot`. Details will be provided when we discuss the formalisation of metric spaces.

5 Formalising Cauchy-Schwarz

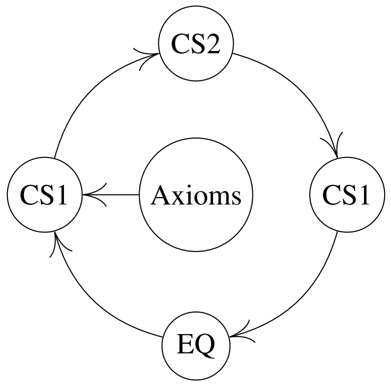
In this section we outline some of the key lemmas in ACL2(r) that result in the Cauchy-Schwarz inequality. Much of the proof is user guided via the algebraic properties of norms, the dot product, etc. Note that the lemma numbers correspond to those in the book `cauchy-schwarz.lisp`. By observing gaps in the numbering sequence in this presentation, the reader can infer where a handful of extra lemmas were needed to complete the proof in ACL2(r).

Program 4.2 An example of using commutativity to prove bilinearity of the dot product.

```
(defthm dot-commutativity
 (implies (and (real-listp vec1) (real-listp vec2)
               (= (len vec2) (len vec1)))
          (= (dot vec1 vec2) (dot vec2 vec1)))
 :hints (("GOAL" :in-theory (enable dot))))

(defthm dot-linear-first-coordinate-1
 (implies (and (real-listp vec1) (real-listp vec2)
               (= (len vec2) (len vec1)) (realp a))
          (= (dot (scalar-* a vec1) vec2)
             (* a (dot vec1 vec2))))
 :hints (("GOAL" :in-theory (enable dot scalar-*))))
...
(defthm dot-linear-second-coordinate-1
 (implies (and (real-listp vec1) (real-listp vec2)
               (= (len vec2) (len vec1)) (realp a))
          (= (dot vec1 (scalar-* a vec2))
             (* a (dot vec1 vec2))))
 :hints (("GOAL" :do-not-induct t
                 :use ((:instance scalar-*-closure (vec vec2))
                       (:instance dot-commutativity (vec2 (scalar-* a vec2)))))))
```

Figure 1: Structure of the proof for Cauchy-Schwarz. CS1, CS2, and EQ denotes (Cauchy-Schwarz I), (Cauchy-Schwarz II), and the conditions for equality, respectively.



5.1 Axioms $\Longrightarrow$ (Cauchy-Schwarz I)

Suppose $v \neq 0$. First we prove

$$\|u - v\|^2 = \langle u, u\rangle - 2\langle u, v\rangle + \langle v, v\rangle \tag{24}$$

by applying multiple instances of the bilinearity of $\langle -,-\rangle$. This can be seen in Program 5.1. Since $\|u-v\| \geq 0$, replacing v with $\frac{\langle u,v\rangle}{\langle v,v\rangle}v$ in Equation (24) above produces

$$0 \leq \left\| u - \frac{\langle u,v\rangle}{\langle v,v\rangle}v \right\|^2 = \langle u,u\rangle - 2\left\langle u, \frac{\langle u,v\rangle}{\langle v,v\rangle}v \right\rangle + \left\langle \frac{\langle u,v\rangle}{\langle v,v\rangle}v, \frac{\langle u,v\rangle}{\langle v,v\rangle}v \right\rangle. \tag{25}$$

This can be seen in Program 5.2 and the inequality reduces to

$$0 \leq \langle u,u\rangle - 2\frac{\langle u,v\rangle}{\langle v,v\rangle}\langle u,v\rangle + \frac{\langle u,v\rangle^2}{\langle v,v\rangle^2}\langle v,v\rangle = \langle u,u\rangle + \langle u,v\rangle\left(-2\frac{\langle u,v\rangle}{\langle v,v\rangle} + \frac{\langle u,v\rangle}{\langle v,v\rangle}\right) = \langle u,u\rangle - \frac{\langle u,v\rangle}{\langle v,v\rangle} \tag{26}$$

Rearranging then produces (Cauchy-Schwarz I). Splitting into the cases $v \neq 0$ and $v = 0$ produces (Cauchy-Schwarz I) for arbitrary vectors u, v since the case where v is zero is trivial. This first version of Cauchy-Schwarz can be seen in Program 5.3.

Program 5.1 Applying bilinearity of the dot product to prove a simple identity.

```
;; < u - v , u - v > = < u , u > - < u , v > - < v , u > + < v , v >
(defthm lemma-3
 (implies (and (real-listp u) (real-listp v) (= (len u) (len v)))
          (equal (norm^2 (vec-- u v))
                 (+ (dot u u) (- (dot u v)) (- (dot v u)) (dot v v))))
 :hints (("GOAL" :use (...(:instance dot-linear-second-coordinate-2
                                     (vec1 v) (vec2 u)
                                     (vec3 (scalar-* -1 v)))
                          (:instance dot-linear-second-coordinate-2
                                     (vec1 u) (vec2 u)
                                     (vec3 (scalar-* -1 v))))))))

;; < u - v, u - v > = < u, u > - 2 < u , v > + < v, v >
(defthm lemma-4
 (implies (and (real-listp u) (real-listp v) (= (len u) (len v)))
          (equal (norm^2 (vec-- u v))
                 (+ (dot u u) (- (* 2 (dot u v))) (dot v v))))
 :hints (("GOAL" :use ((:instance dot-commutativity (vec1 u) (vec2 v))))))
```

5.2 (Cauchy-Schwarz I) $\Longleftrightarrow$ (Cauchy-Schwarz II)

To see (Cauchy-Schwarz II) from (Cauchy-Schwarz I), we simply take square roots and show the equivalence between the dot products and the square of the norms. This part can be seen in Program 5.4. To see the other direction, we simply square both sides and rearrange.

5.3 (Cauchy-Schwarz I) $\Longleftrightarrow$ Conditions for Equality

Suppose $u, v \neq 0$ and

$$\langle u,v\rangle^2 = \langle u,u\rangle\langle v,v\rangle. \tag{27}$$

Program 5.2 Substituting v for $\langle v,v\rangle^{-1}\langle u,v\rangle v$.

```
;; 0 <= < u, u > - 2 < u , v > + < v, v >
(local (defthm lemma-6
 (implies (and (real-listp u) (real-listp v) (= (len u) (len v)))
          (equal (<= 0 (norm^2 (vec-- u v)))
                 (<= 0 (+ (dot u u) (- (* 2 (dot u v))) (dot v v)))))))

;; let v = (scalar-* (* (/ (dot v v)) (dot u v)) v)
(local (defthm lemma-7
 (implies (and (real-listp u) (real-listp v) (= (len u) (len v)))
          (equal (<= 0 (norm^2 (vec-- u (scalar-* (* (/ (dot v v)) (dot u v)) v))))
                 (<= 0 (+ (dot u u)
                          (- (* 2 (dot u (scalar-* (* (/ (dot v v)) (dot u v)) v))))
                          (dot (scalar-* (* (/ (dot v v)) (dot u v)) v)
                               (scalar-* (* (/ (dot v v)) (dot u v)) v))))))
 :hints (("GOAL" :use (...(:instance lemma-6 (v (scalar-* (* (/ (dot v v))
                                                            (dot u v)) v))))))))
```

Program 5.3 Final form of (Cauchy-Schwarz I).

```
(defthm cauchy-schwarz-1
 (implies (and (real-listp u) (real-listp v) (= (len u) (len v)))
                 (<= (* (dot u v) (dot u v))
                     (* (dot u u) (dot v v))))
 :hints (("GOAL" ... :cases ((zvecp v) (not (zvecp v))))))
```

Then we simply reverse all the equalities used to prove (Cauchy-Schwarz I) from the axioms (see Inequality (25)) until we return to

$$0 = \left\| u - \frac{\langle u,v\rangle}{\langle v,v\rangle} v \right\|^2. \tag{28}$$

Since $\|\cdot\|^2$ is positive definite, we must have

$$u = \frac{\langle u,v\rangle}{\langle v,v\rangle} v. \tag{29}$$

This can be seen in Program 5.5.

To show linearity, we introduce a Skolem function so that the final form of the conditions for equality are in the greatest generality. We also attempted a proof where the value of the Skolem constant was explicitly computed – simply find the first non-zero element of v and divide the corresponding element of u by the v element. The proof for the Skolem function approach was much simpler because the witness value comes already endowed with the properties we need for subsequent reasoning. In particular, invoking linear dependence to show (Cauchy-Schwarz I) simply amounts to applying algebraic rules to an arbitrary unknown witness which is simple (though occasionally tedious) from our formalisation. Otherwise, not using a Skolem function would necessitate the exhibition of a particular coefficient to complete the implication graph in Figure 1. The definition of the Skolem function is in Program 5.6. The cases for

Program 5.4 Showing (Cauchy-Schwarz II).

```
(local (defthm lemma-16
 (implies (and (real-listp u) (real-listp v) (= (len u) (len v)))
          (and (equal (acl2-sqrt (* (dot u v) (dot u v))) (abs (dot u v)))
               (equal (acl2-sqrt (dot u u)) (eu-norm u))
               (equal (acl2-sqrt (dot v v)) (eu-norm v))
               (equal (acl2-sqrt (* (dot u u) (dot v v)))
                      (* (eu-norm u) (eu-norm v)))))
 :hints (("GOAL" :use ((:instance norm-inner-product-equivalence (vec u))
                       (:instance norm-inner-product-equivalence (vec v)))))))

(defthm cauchy-schwarz-2
 (implies (and (real-listp u) (real-listp v) (= (len u) (len v)))
          (<= (abs (dot u v))
              (* (eu-norm u) (eu-norm v))))
 :hints (("GOAL" :use ((:instance cauchy-schwarz-1)
                       (:instance norm-inner-product-equivalence (vec v))
                       (:instance norm-inner-product-equivalence (vec u)) ...) ...)))
```

$u = 0$ or $v = 0$ are immediate. The final result can be seen in Program 5.7. The conditions for equality for (Cauchy-Schwarz II) follows from its equivalence to (Cauchy-Schwarz I).

Program 5.5 Linearity follows from equality.

```
(local (defthm lemma-19
 (implies (and (real-listp u) (real-listp v) (= (len u) (len v)) (not (zvecp v))
               (= (* (dot u v) (dot u v)) (* (dot u u) (dot v v))))
          (equal u (scalar-* (* (/ (dot v v)) (dot u v)) v))) ...))
```

Program 5.6 Introducing a Skolem function for linearity.

```
(defun-sk linear-dependence-nz (u v)
 (exists a (equal u (scalar-* a v))))
```

5.4 Final Statement of the Cauchy-Schwarz Inequality

Program 5.8 displays our final form of Cauchy-Schwarz. The ACL2(r) theorems `cauchy-schwarz-1` and `cauchy-schwarz-2` are equivalent to (Cauchy-Schwarz I) and (Cauchy-Schwarz II), respectively. The theorems `cauchy-schwarz-3` and `cauchy-schwarz-4` correspond to the conditions for equality for (Cauchy-Schwarz I) and (Cauchy-Schwarz II), respectively.

Program 5.7 The conditions for equality for (Cauchy-Schwarz I).

```
(defthm cauchy-schwarz-3
 (implies (and (real-listp u) (real-listp v) (= (len u) (len v)))
          (equal (= (* (dot u v) (dot u v)) (* (dot u u) (dot v v)))
                 (or (zvecp u) (zvecp v) (linear-dependence-nz u v)))) ...)
```

Program 5.8 The final form of Cauchy-Schwarz.

```
(defthm cauchy-schwarz-1
 (implies (and (real-listp u) (real-listp v) (= (len u) (len v)))
                 (<= (* (dot u v) (dot u v))
                     (* (dot u u) (dot v v)))) ...)

(defthm cauchy-schwarz-2
 (implies (and (real-listp u) (real-listp v) (= (len u) (len v)))
          (<= (abs (dot u v))
              (* (eu-norm u) (eu-norm v)))) ...)

(defthm cauchy-schwarz-3
 (implies (and (real-listp u) (real-listp v) (= (len u) (len v)))
          (equal (= (* (dot u v) (dot u v))
                    (* (dot u u) (dot v v)))
                 (or (zvecp u) (zvecp v)
                     (linear-dependence-nz u v)))) ...)

(defthm cauchy-schwarz-4
 (implies (and (real-listp u) (real-listp v) (= (len u) (len v)))
          (equal (=  (abs (dot u v)) (* (eu-norm u) (eu-norm v)))
                 (or (zvecp u) (zvecp v)
                     (linear-dependence-nz u v)))) ...)
```

6 $\mathbb{R}^n$ as a Metric Space in ACL2(r)

6.1 Metric Space Axioms

Observe

$$d_2(x,y) = \|x-y\|_2 = \sqrt{\langle x-y, x-y\rangle}. \tag{30}$$

Proving theorems regarding metrics reduces to proving theorems about the norm from which the metric is induced. Likewise, proving theorems involving norms can be reduced to proving properties about the underlying inner product. The process of formalisation, then, should ideally define the metric via the norm, and the norm should be defined via the inner product. This is useful for proving properties such as positive definiteness of both the metric and the norm, eg. if

$$\langle u,u\rangle \geq 0 \tag{31}$$

with equality iff $u = 0$, then the same applies for

$$\|x\|_2 = \sqrt{\langle x,x\rangle} \geq 0 \tag{32}$$

and so

$$d_2(x,y) = \|x-y\|_2 = \sqrt{\langle x-y, x-y\rangle} \geq 0. \tag{33}$$

However, as exemplified by `vec--` and `vec--x`, the obvious sequence is not necessarily the easiest. Indeed, not only is it simpler to prove theorems on functions equivalent to the desired functions, we also prove properties of similar functions not equivalent to the desired functions but such that if the properties hold for the similar functions, then they also hold for the desired functions. For example, suppose we wish to prove commutativity for the Euclidean metric `eu-metric`. Recall `(vec-- x y)` is a macro for `(vec-+ x (scalar-* -1 y))` and, together with an instance of `acl2-sqrt`, formalising the proofs for commutativity require a non-trivial amount of hints and user guidance. We instead define recursively a function `metric^2`, which is the square of the norm of the difference of two vectors (which is equivalent to the square of the Euclidean metric, i.e. $\|x-y\|_2^2$). Moreover, proving those equivalences simply amounts to unwinding the definitions. Having established

$$\|x-y\|_2^2 = \|y-x\|_2^2, \tag{34}$$

it follows that

$$d_2(x,y) = \sqrt{\|x-y\|_2^2} = \sqrt{\|y-x\|_2^2} = d_2(y,x). \tag{35}$$

Hence, commutativity for `eu-metric` is proven.

Program 6.1 The Euclidean norm, metric, and squared metric.

```
(defun eu-norm (u)
 (acl2-sqrt (dot u u)))
...
(defun eu-metric (u v)
 (eu-norm (vec-- u v)))
...
(define metric^2 (vec1 vec2) ...
 (norm^2 (vec-- vec1 vec2)) ...)
```

6.2 Continuity $\mathbb{R}^n \to \mathbb{R}$

To showcase continuity, let us begin with an enlightening example. Recall the non-standard analysis definition of continuity for a function $f : \mathbb{R}^n \to \mathbb{R}$ stated in the language of non-standard analysis: if $d_2(x,y)$ is an infinitesimal for a standard x, then so is $f(x) - f(y)$. Take $f(x) = \sum_{i=1}^n x_i$. It is clear that f is continuous in our usual theory of classical real analysis. However, we must translate this into the language of infinitesimals. By hypothesis,

$$d_2(x,y) = \|x-y\|_2 = \sqrt{(x_1-y_1)^2 + (x_2-y_2)^2 + \cdots + (x_n-y_n)^2} \tag{36}$$

is an infinitesimal. We would like to show that

$$f(x) - f(y) = \sum_{i=1}^n x_i - \sum_{i=1}^n y_i = \sum_{i=1}^n (x_i - y_i) \tag{37}$$

is also an infinitesimal. Indeed, by Equation (36) we see that each $x_i - y_i$ must necessarily be infinitesimal since otherwise $d_2(x,y)$ wouldn't be an infinitesimal. Because the RHS of Equation (37) is a finite sum of infinitesimals, so is $f(x) - f(y)$ as desired. The two motivating questions are:

1. How do we make ACL2(r) recognize $x_i - y_i$ are infinitesimals from $d_2(x,y)$ being infinitesimal?
2. How do we state "all $x_i - y_i$ are infinitesimals"?

To answer the first question, observe for any vector z and $i \leq n$,

$$\|z\|_2 = \sqrt{\sum_{i=1}^{n} z_i^2} \geq \max_i |z_i| \geq |z_i|. \tag{38}$$

Setting $z = x - y$, we see that if the norm is an infinitesimal, then so must each entry of the vector. By introducing an ACL2(r) function, say `max-abs-reals`, equivalent to $\max_i$ and reasoning over the arbitrariness of i instead of over the length of x and y, we may exhibit the infinitesimality of any entry in $x - y$ as seen in Program 6.2.

Program 6.2 Showing arbitrary entries of a vector are infinitesimal via the maximum element.

```
(define max-abs-reals ((vec real-listp))
 ...
 (b* (((unless (consp vec)) 0)
       ((cons hd tl) vec)
       ((unless (realp hd)) 0))
      (max (abs hd) (max-abs-reals tl)))...)

(defthm eu-norm-i-small-implies-max-abs-reals-i-small
 (implies (and (real-listp vec) (i-small (eu-norm vec)))
          (i-small (max-abs-reals vec))))

(defthm eu-norm-i-small-implies-elements-i-small
 (implies (and (real-listp vec) (i-small (eu-norm vec)) (natp i) (< i (len vec)))
          (i-small (nth i vec))))
```

To address the second question, one could imagine a recognizer for vectors with infinitesimal entries – such a recognizer is depicted in Program 6.3. However, this recognizer would be recursive on the entries of the vector with each recursive step invoking the non-classical recognizer `i-small` for infinitesimal reals. Because non-classical recursive functions are forbidden, so is the suggested recognizer. A Skolem function was also considered as a possibility but to remain consistent with `eu-norm-i-small-implies-elements-i-small` a theorem positing the condition for an arbitrary index i as seen in Program 6.4 was chosen instead.

Program 6.3 A fantastical recognizer for vectors with infinitesimal entries that doesn't exist.

```
(defun i-small-vecp (x)
 (cond ((null x) t)
       ((not (real-listp x)) nil)
       (t (and (i-small (car x)) (i-small-vecp (cdr x))))))
```

To see `eu-metric-i-small-implies-difference-of-entries-i-small` in action, consider once again the example of $f(x) = \sum_{i=1}^{n} x_i$. If $n = 3$ and `sum` is the ACL2(r) function equivalent to f, then the following is the proof of continuity for `sum`. Other examples of functions with proofs of continuity

Program 6.4 A theorem positing the infinitesimality of arbitrary entries in $x - y$.

```
(defthm eu-metric-i-small-implies-difference-of-entries-i-small
 (implies (and (real-listp x) (real-listp y) (= (len y) (len x))
               (natp i) (< i (len x)))
               (i-small (eu-metric x y))
          (i-small (- (nth i x) (nth i y)))) ...)
```

in ACL2(r) include the Euclidean norm, the dot product with one coordinate fixed, and the function $g(x,y) = xy$.

Program 6.5 A theorem positing `sum` is continuous.

```
(defthm sum-is-continuous
 (implies (and (real-listp x) (real-listp y) (= (len x) 3) (= (len y) (len x))
               (i-small (eu-metric x y)))
          (i-small (- (sum x) (sum y))))...)
```

7 Conclusion

Firstly, we would like to note the choice of classical proof on which we base this formalisation. In particular, we would like to compare its flavour to other proofs of Cauchy-Schwarz. Indeed, there are geometric proofs, analytical proofs, combinatorial proofs, inductive proofs, etc. [25] whereas we followed a rather algebraic approach. Considering ACL2(r)'s strengths with regards to induction, the choice may seem odd. Indeed, there are several potential inductive candidates we considered at the onset of this endeavor before proceeding with the algebraic approach. However, most of the other candidates inducted over the dimension of $\mathbb{R}^n$ and required reasoning over the real entries of vectors. We suspect unwinding the vectors and guiding ACL2(r) through such a proof would be more onerous than the one outlined in this paper. Moreover, our formalisation of inner product spaces already provided the exact tools necessary for our preferred proof of Cauchy-Schwarz (eg. vectors, vector-vector operations, scalar-vector operations, inner products, etc.) without resorting to reasoning over individual reals. The precision of this approach, while arguably more elegant, also complements our approach to defining continuity. By reasoning over the properties of the vector itself instead of the individual components, we circumvent the soundness-motivated limitations of ACL2(r).

Secondly, this formalisation has two notable purposes. The first is the various applications that may be introduced as a result of the Cauchy-Schwarz inequality. The appearance of Cauchy-Schwarz in functional analysis, real analysis, probability theory, combinatorics, etc. speaks to its utility. A further application of Cauchy-Schwarz is in [10] where a set of theorems involving convex functions are formalised. Additionally, the various structures of $\mathbb{R}^n$ are very rich in mathematical theory and hold applications in various areas of science. In this paper, we presented a formalisation of the space from only two perspectives. However, the choice of perspectives is arguably among the most fundamental. It is the vector space structure of $\mathbb{R}^n$ that provides the necessary operations between its elements. Indeed, one would be hard-pressed to find any view of $\mathbb{R}^n$ that does not assume operations on the domain. Moreover, inner products are the path to calculus: inner products lead to norms; norms lead to metrics; metrics lead to real analysis. The formalisation of $\mathbb{R}^n$ as a metric space is the last step before multivariate calculus, which is in of itself highly applicable and left as future work. We also note the possibility of proving Cauchy-Schwarz for more abstract structures as future work.

During the course of formalisation, emphasis was placed on using algebraic methods to prove theorems that would have otherwise been proved via induction. However, algebraic approaches require significant guidance from the user. Instead, the properties of the inner product space axioms and the proof of Cauchy-Schwarz might be amenable to certification by a SMT solver via `smtlink` [16, 15]. The challenge here is that SMT solvers do not perform induction – we need to leave that for ACL2. On the other hand, we might be able to treat operations on vectors as uninterpreted functions with constraints corresponding to the requirements for a function to be an inner product, a norm, etc.

Finally, we discuss further formalisations of $\mathbb{R}^n$ under different lenses. In fact, there is still potential to further extend $\mathbb{R}^n$ as a metric space. The notions of continuity are independent of the metric used and d_2 may be replaced with any metric on $\mathbb{R}^n$. By way of encapsulation, pseudo-higher-order techniques may be employed to easily formalise various real metric spaces – especially if we consider the metric induced by other p-norms.

Among the extensions of $\mathbb{R}^n$ as a metric space is proving its completeness. Addressing Cauchy sequences traditionally follows from an application of Bolzano-Weierstrass which has yet to be formalised in ACL2 [20]. Stating completeness in terms infinitesimals and ACL2(r) is a farther but tantalizing prospect. Upon doing so, we would have a formalisation of $\mathbb{R}^n$ as a Hilbert space [17].

References

[1] ACL2: *A Beginners Guide to Reasoning about Quantification in ACL2*. Available at `https://www.cs.utexas.edu/users/moore/acl2/current/manual/index.html?topic=ACL2___ _QUANTIFIER-TUTORIAL`. Originally written by Sandip Ray.

[2] Sanaz Khan Afshar, Vincent Aravantinos, Osman Hasan & Sofiène Tahar (2014): *Formalization of Complex Vectors in Higher-Order Logic*. In Stephen M. Watt, James H. Davenport, Alan P. Sexton, Petr Sojka & Josef Urban, editors: *Intelligent Computer Mathematics*, Springer International Publishing, Cham, pp. 123–137, doi:10.1023/A:1012692601098.

[3] Leif O. Arkeryd, Nigel J. Cutland & C. Ward Henson (Eds.) (1997): *Nonstandard Analysis: Theory and Applications*, 1st edition. *Nato Science Series C: 493* , Springer Netherlands, doi:10.1007/978-94-011-5544-1.

[4] John Cowles & Ruben Gamboa (2017): *The Cayley-Dickson Construction in ACL2*. In Anna Slobodova & Warren Hunt, Jr., editors: Proceedings 14th International Workshop on the *ACL2 Theorem Prover and its Applications*, Austin, Texas, USA, May 22-23, 2017, *Electronic Proceedings in Theoretical Computer Science* 249, Open Publishing Association, pp. 18–29, doi:10.4204/EPTCS.249.2.

[5] Ruben A. Gamboa & Matt Kaufmann (2001): *Nonstandard Analysis in ACL2*. *Journal of Automated Reasoning* 27(4), pp. 323–351, doi:10.1023/A:1011908113514.

[6] John Harrison (2005): *A HOL Theory of Euclidean Space*. In Joe Hurd & Tom Melham, editors: *Theorem Proving in Higher Order Logics*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 114–129, doi:10.1007/11541868_8.

[7] Nathan Jacobson (1985): *Basic Algebra I*, 2nd edition. Dover Publications.

[8] Nathan Kahl: *The Hundred Greatest Theorems*. Online. Available at `http://pirate.shu.edu/~kahlnath/Top100.html`. Originally published by Paul and Jack Abad (1999).

[9] Gerwin Klein: *The Top 100 Theorems in Isabelle*. Online. Available at `https://www.cse.unsw.edu.au/~kleing/top100/#78`.

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

[11] Serge Lang (2002): *Algebra*, 3rd edition. *Graduate Texts in Mathematics 211* , Springer-Verlag New York, doi:10.1007/978-1-4613-0041-0.

[12] Peter A. Loeb & Manfred P. H. Wolff (2015): *Nonstandard Analysis for the Working Mathematician*, 2nd edition. Springer Netherlands, doi:10.1007/978-94-017-7327-0.

[13] Jean-Marie Madiot: *Formalizing 100 theorems in Coq*. Online. Available at `https://madiot.fr/coq100/#78`.

[14] Marco Maggesi (2018): *A Formalization of Metric Spaces in HOL Light*. *Journal of Automated Reasoning* 60(2), pp. 237–254, doi:10.1007/s10817-017-9412-x.

[15] Yan Peng & Mark Greenstreet (2015): *Integrating SMT with Theorem Proving for Analog/Mixed-Signal Circuit Verification*. In Klaus Havelund, Gerard Holzmann & Rajeev Joshi, editors: *NASA Formal Methods*, Springer International Publishing, Cham, pp. 310–326, doi:10.1007/978-3-319-17524-9_22.

[16] Yan Peng & Mark R. Greenstreet (2015): *Extending ACL2 with SMT Solvers*. In: *Proceedings Thirteenth International Workshop on the ACL2 Theorem Prover and Its Applications, Austin, Texas, USA, 1-2 October 2015.*, pp. 61–77, doi:10.4204/EPTCS.192.6.

[17] Frigyes Riesz & Bela Sz.-Nagy (1990): *Functional Analysis*. Dover Publications.

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

[19] Steven Roman (2008): *Advanced Linear Algebra*, 3rd edition. *Graduate Texts in Mathematics 135* , Springer-Verlag New York, doi:10.1007/978-0-387-72831-5.

[20] Walter Rudin (1976): *Principles of Mathematical Analysis*, 3rd edition. *International Series in Pure and Applied Mathematics* , McGraw-Hill.

[21] Georgi E. Shilov (1977): *Linear Algebra*. Dover Publications.

[22] J. Michael Steele (2004): *The Cauchy-Schwarz Master Class: An Introduction to the Art of Mathematical Inequalities*. Cambridge University Press, doi:10.1017/CBO9780511817106.

[23] Jasper Stein (2001): *Documentation for the formalization of Linerar Agebra*. Online. Available at `http://www.cs.ru.nl/~jasper/`.

[24] Freek Wiedijk: *Formalizing 100 Theorems*. Online. Available at `http://www.cs.ru.nl/~freek/100`.

[25] Hui-Hua Wu & Shanhe Wu (2009): *Various proofs of the Cauchy-Schwarz inequality*. *Octogon Mathematical Magazine* 17(1), pp. 221–229.

Appendix A Why are Non-classical Recursive Functions Prohibited?

A discussion with Ruben Gamboa[1] and members of the ACL2 Help Mailing List sheds light on why non-classical recursive functions are prohibited. In summary, the introduction of such functions will also introduce inconsistency into the logic of ACL2(r). It is possible to define a function that violates the rules of non-standard analysis.

For example, consider the hypothetical function defined in Program A.1. Suppose `f` terminates. If `n` is standard, then `f` returns it without issue. If `n` is infinite, then `f` returns the largest standard number. However, this is impossible since such a number does not exist and, if `n` is infinite, so is `(-1 n)` which means `f` should not have terminated anyways. Moreover, this applies if `n` is any non-standard hyperreal

[1] We would like to thank Ruben Gamboa for his insightful explanations on which most of this appendix is based.

Program A.1 An impossible function in ACL2(r).

```
(defun f (n)
 (cond ((zp n) 0)
       ((standardp n) n)
       (t (f (-1 n)))))
```

since if $n = \mathrm{st}(n) + \varepsilon$ is equivalent to `n` and $\varepsilon > 0$ is an infinitesimal, then $n - 1 = \mathrm{st}(n-1) + \varepsilon$ is not standard either.

The essence of the issue with `f` is that its measure is non-standard. If the measure of a function can be proven to be standard, then a recursive non-classical function could be conceded. However, in practice, such a proof would likely be subtle and involved.

Convex Functions in ACL2(r)

Carl Kwan   Mark R. Greenstreet

Department of Computer Science
University of British Columbia[*]
Vancouver, Canada

{carlkwan, mrg}@cs.ubc.ca

This paper builds upon our prior formalisation of $\mathbb{R}^n$ in ACL2(r) by presenting a set of theorems for reasoning about convex functions. This is a demonstration of the higher-dimensional analytical reasoning possible in our metric space formalisation of $\mathbb{R}^n$. Among the introduced theorems is a set of equivalent conditions for convex functions with Lipschitz continuous gradients from Yurii Nesterov's classic text on convex optimisation. To the best of our knowledge a full proof of the theorem has yet to be published in a single piece of literature. We also explore "proof engineering" issues, such as how to state Nesterov's theorem in a manner that is both clear and useful.

1 Introduction

Convex optimisation is a branch of applied mathematics that finds widespread use in financial modelling, operations research, machine learning, and many other fields. Algorithms for convex optimisation often have many parameters that can be tuned to improve performance. However, a choice of parameter values that produces good performance on a set of test cases may suffer from poor convergence or non-convergence in other cases. Hand written proofs for convergence properties often include simplifying assumptions to make the reasoning tractable. This motivates using machine generated and/or verified proofs for the convergence and performance of these algorithms. Once an initial proof has been completed, the hope is that simplifying assumptions can be incrementally relaxed or removed to justify progressively more aggressive implementations. These observations motivate our exploration of convex functions within ACL2(r).

We present example proofs of continuity, Lipschitz continuity, and convexity for some simple functions as well as some basic theorems of convex optimisation. To the best of our knowledge, there are no other formalisations of convex functions in published literature (though we were able to find some formal theorems involving convex hulls in [5]). Moreover, we also provide a proof for a set of equivalent conditions for inclusion in the class of convex functions with Lipschitz continuous gradients – a theorem that, to the best of our knowledge, has yet to be fully published in a single piece of literature with a correct proof. This characterisation is based on Yurii Nesterov's classic work on convex optimisation [11] which has applications in the convergence proofs for many gradient descent algorithms.

This paper builds on the formalisation of $\mathbb{R}^n$ as an inner product space and as a metric space in [8]. Of particular note, we make use of the Cauchy-Schwarz inequality which was a demonstration of the algebraic reasoning capable in such a formalisation. This subsequent paper demonstrates the analytical reasoning about multivariate functions $\mathbb{R}^n \to \mathbb{R}$ that is enabled by the theorems proven in the books from the previous paper. In addition to presenting some key lemmas, we also discuss some of the challenges of formalising theorems with proofs that rely heavily on informally well-established and intuitive notions.

[*]This work is supported in part by the National Science and Engineering Research Council of Canada (NSERC) Discovery Grant program and the Institute for Computing, Information and Cognitive Systems (ICICS) at UBC.

2 Preliminaries

This section summarizes the formalisation of vector spaces, inner-product spaces, metric spaces, and the Cauchy-Schwarz inequality that are presented in our previous paper [8]. The ACL2(r) formalisation is provided in the ACL2 books that accompany these papers. This work depends greatly on the original formalisation of the reals via non-standard analysis in ACL2(r) [4]. Further background on non-standard analysis can be found in [12, 1, 10]. Background on vector, inner product, and metric spaces can be found in [15, 13, 14, 9, 6]. We also outline some theorems involving convex functions. Standard texts on convex optimisation include [2, 11].

2.1 Inner Product & Metric Spaces

An inner product space is a vector space (V,F) equipped with an inner product $\langle -,- \rangle : V \to F$. The inner product satisfies

$$a\langle u,v\rangle = \langle au,v\rangle \tag{1}$$
$$\langle u+v,w\rangle = \langle u,w\rangle + \langle v,w\rangle \tag{2}$$
$$\langle u,v\rangle = \langle v,u\rangle \text{ when } F=\mathbb{R} \tag{3}$$
$$\langle u,u\rangle \geq 0 \text{ with equality iff } u=0 \tag{4}$$

for any $u,v,w \in V$ and $a \in F$ [13, Chapter 9].

An inner product induces a norm $\|\cdot\|$. If $\|\cdot\| = \sqrt{\langle -,- \rangle}$, then the Cauchy-Schwarz inequality [9, Chapter 15] holds for any $x,y \in V$:

$$|\langle x,y\rangle| \leq \|x\|\|y\|. \tag{5}$$

Norms induce metrics [14, Chapter 2] $d(x,y) = \|x-y\|$ which satisfy

$$d(x,y)=0 \iff x=y \qquad \textit{(definitness)} \tag{6}$$
$$d(x,y)=d(y,x) \qquad \textit{(symmetry)} \tag{7}$$
$$d(x,y) \leq d(x,z)+d(z,y) \qquad \textit{(triangle inequality).} \tag{8}$$

From this it follows that

$$d(x,y) \geq 0 \tag{9}$$

because

$$0 = d(x,x) \leq d(x,y)+d(y,x) = 2d(x,y).$$

A metric space is a pair (M,d) where M is a set and d is a metric on M [14, Chapter 2].

A function $f : M \to M'$ between metric spaces is continuous [14, Chapter 4] if for every $x \in M$ and for any $\varepsilon > 0$, there is a $\delta > 0$ such that

$$d_M(x,y) < \delta \implies d_{M'}(f(x),f(y)) < \varepsilon \tag{10}$$

for any $y \in E$.

Throughout this paper, we will consider $\mathbb{R}^n$ adjoined with the dot product and Euclidean metric to be an inner product and metric space, respectively.

2.2 Continuity & Differentiability

A univariate function $f : \mathbb{R} \to \mathbb{R}$ is continuous if for any $x \in \mathbb{R}$ and $\varepsilon > 0$, there is a $\delta > 0$ such that for any $y \in \mathbb{R}$, if $|x-y| < \delta$, then $|f(x) - f(y)| < \varepsilon$ [14, Chapter 4]. Moreover, the derivative of f is defined to be

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{11}$$

if such a form exists [14, Chapter 5].

For a multivariate function $f : \mathbb{R}^n \to \mathbb{R}$, continuity is defined similarly. We call f continuous if for any $x \in \mathbb{R}^n$ and $\varepsilon > 0$, there is a $\delta > 0$ such that for any $y \in \mathbb{R}^n$, if $\|x-y\| < \delta$, then $|f(x) - f(y)| < \varepsilon$ [14, Chapter 4]. Likewise, if it exists, there is a derivative for multivariate functions defined similarly to the univariate case:

$$\langle f'(x), h\rangle = \lim_{\|h\|_2 \to 0} \frac{f(x+h) - f(x)}{\|h\|_2}. \tag{12}$$

We call $f' : \mathbb{R}^n \to \mathbb{R}^n$ the *gradient* of f and it satisfies

$$f'(x) = (f_1'(x_1), f_2'(x_2), \ldots, f_n'(x_n)) \tag{13}$$

where $f_i'(x_i)$ is the univariate derivative of f with respect to the i-th component x_i of x [14, Chapter 9].

2.3 Convex Functions & $\mathscr{F}_L^1(\mathbb{R}^n)$

A function $f : \mathbb{R}^n \to \mathbb{R}$ is convex [2, Chapter 3] if for any $x, y \in \mathbb{R}^n$ and $\alpha \in [0,1]$,

$$\alpha f(x) + (1-\alpha) f(y) \geq f(\alpha x + (1-\alpha) y). \tag{14}$$

Equivalently [11, Def. 2.1.1], if f is differentiable once with gradient f', then it is convex if

$$f(y) \geq f(x) + \langle f'(x), y-x\rangle. \tag{15}$$

Following Nesterov, we write $\mathscr{F}(\mathbb{R}^n)$ to denote the class of convex functions from $\mathbb{R}^n$ to $\mathbb{R}$. Examples of convex functions include $f(x) = x^2$, $\|\cdot\|_2$, and $\|\cdot\|_2^2$. Moreover, the class of convex functions is closed under certain operations [2, Chapter 3].

Theorem 1. If $f : \mathbb{R}^n \to \mathbb{R}$ and $g : \mathbb{R}^n \to \mathbb{R}$ and $h : \mathbb{R} \to \mathbb{R}$ are convex with h monotonically increasing, then

1. $a \cdot f$ is convex for any real $a \geq 0$,
2. $f + g$ is convex,
3. $h \circ f$ is convex.

Informal proofs of these claims follow from the definitions and can be found in [2, Chapter 3].

Often, convex optimisation algorithms require f to be both convex and sufficiently "smooth". Here, we take "smooth" to be stronger than continuous but not necessarily differentiable. In particular, we say that $f : \mathbb{R}^n \to \mathbb{R}$ is Lipschitz continuous if for any $x, y \in \mathbb{R}^n$ there is some $L > 0$ such that

$$\|f(x) - f(y)\| \leq L\|x-y\|. \tag{16}$$

Informally, we have the following chain of inclusions for classes of functions:

$$\text{Differentiable} \subset \text{Lipschitz Continuous} \subset \text{Continuous}.$$

We write $\mathscr{F}_L^1(\mathbb{R}^n)$ for the class of convex differentiable functions on $\mathbb{R}^n$ with Lipschitz continuous gradient with constant L. Functions in the class $\mathscr{F}_L^1(\mathbb{R}^n)$ have many useful properties for optimisation. The main result of this paper is proving a theorem from [11, Thm. 2.1.5] that gives six "equivalent" ways of showing that a convex function is in $\mathscr{F}_L^1(\mathbb{R}^n)$:

Theorem 2 (Nesterov)**.** Let $f \in \mathscr{F}^1(\mathbb{R}^n)$, $x,y \in \mathbb{R}^n$ and $\alpha \in [0,1]$. The following conditions are equivalent to $f \in \mathscr{F}_L^1(\mathbb{R}^n)$:

$$f(y) \le f(x) + \langle f'(x), y-x\rangle + \frac{L}{2}\|x-y\|^2 \tag{Nest. 1}$$

$$f(x) + \langle f'(x), y-x\rangle + \frac{1}{2L}\|f'(x)-f'(y)\|^2 \le f(y) \tag{Nest. 2}$$

$$\frac{1}{L}\|f'(x)-f'(y)\|^2 \le \langle f'(x)-f'(y), x-y\rangle \tag{Nest. 3}$$

$$\langle f'(x)-f'(y), x-y\rangle \le L\|x-y\|^2 \tag{Nest. 4}$$

$$f(\alpha x + (1-\alpha)y) + \frac{\alpha(1-\alpha)}{2L}\|f'(x)-f'(y)\|^2 \le \alpha f(x) + (1-\alpha)f(y) \tag{Nest. 5}$$

$$\alpha f(x) + (1-\alpha)f(y) \le f(\alpha x + (1-\alpha)y) + \alpha(1-\alpha)\frac{L}{2}\|x-y\|^2. \tag{Nest. 6}$$

There are several motivations to prove Thm. 2 in ACL2(r). First, such a formalisation provides an unambiguous statement of the theorem. For example, the theorem requires the assumption $f \in \mathscr{F}^n$, but this hypothesis is not explicitly stated in the theorem statement in [11]. Instead, the assumption is implicit in the preceding text. On the other hand, Nest. 5 implies Eq. 14 and therefore that f is convex. Inequalities Nest. 2 through Nest. 6 implicitly have an existential quantification of L. By stating and proving the theorem in ACL2(r), these ambiguities are avoided. Furthermore, this enables the use of Nesterov's theorem for further reasoning about convex functions and optimisation algorithms.

2.4 ACL2(r) & Non-standard Analysis

The usual axioms for $\mathbb{R}^n$ as an inner product and metric space were formalised in ACL2(r) in [8] along with theorems proving their salient properties. Reals and infinitesimals are recognized by `realp` and `i-small`, respectively. Two reals are `i-close` if their difference is `i-small`. Vectors are recognized by `real-listp`. Vector addition and subtraction are `(vec-+ x y)` and `(vec-- x y)`, respectively. Scalar multiplication is `(scalar-* a x)`. The dot product is simply `(dot x y)`. The Euclidean norm and metric are `(eu-norm x)` and `(eu-metric x y)`, respectively. Sometimes it is easier to reason about the square of the norm or metric. These are called `(norm^2 x)` and `(metric^2 x y)`, respectively. More details can be found in [8].

In non-standard analysis, continuity for a function $f:\mathbb{R}^n \to \mathbb{R}$ amounts to if $\|x-y\|$ is an infinitesimal for some standard x, then so is $|f(x)-f(y)|$. The derivative of a function $f:\mathbb{R}\to\mathbb{R}$ is the standard part of $\frac{f(x+h)-f(x)}{h}$ where h is an infinitesimal. The formalisation for continuity, differentiability, integrability, and the Fundamental Theorem of Calculus already exist for univariate functions in ACL2(r). [3, 7]

3 Convexity in ACL2(r)

In this section, we provide some selected examples of formalised theorems involving convex functions. The formalised proofs follow almost directly from those of the informal proofs. For the sake of exposition, the proof for the first theorem is outlined but the rest are omitted.

The first is a simple theorem positing the convexity of $f(x) = x^2$ which is true because for $0 \leq \alpha \leq 1$,

$$\alpha x^2 + (1-\alpha)y^2 - (\alpha x + (1-\alpha)y)^2 = \alpha(1-\alpha)(x^2 - 2xy + y^2) = \alpha(1-\alpha)(x-y)^2 \geq 0. \qquad (17)$$

We first define a square function: `(defun square-fn (x) (* (realfix x) (realfix x)))`. The chain of equalities in Eq. 17 is immediately recognized by ACL2(r). The inequality in Eq. 17 also passes without issue. The convexity of `square-fn` then follows from a simple application of the two lemmas.

Program 3.1 The equality in Eq. 17 formalised.

```
;; ax^2 + (1-a)y^2 - (ax + (1-a)y)^2 = a(1-a)(x-y)^2
(defthm lemma-1
 (implies (and (realp x) (realp y) (realp a) (<= 0 a) (<= a 1))
          (equal (- (+ (* a (square-fn x)) (* (- 1 a) (square-fn y)))
                    (square-fn (+ (* a x) (* (- 1 a) y))))
                 (* a (- 1 a) (square-fn (- x y))))))
```

Program 3.2 A more general version of the inequality in Eq. 17 formalised.

```
;; replace a with a(1-a) and x with x-y to obtain the desired inequality
(defthm lemma-2
 (implies (and (realp a) (<= 0 a))
          (<= 0 (* a (square-fn x)))))
```

Program 3.3 The square function is convex.

```
(defthm square-fn-is-convex
 (implies (and (realp x) (realp y) (realp a) (<= 0 a) (<= a 1))
          (<= (square-fn (+ (* a x) (* (- 1 a) y)))
              (+ (* a (square-fn x)) (* (- 1 a) (square-fn y)))))
 :hints (("GOAL" :use ((:instance lemma-2 (a (* a (- 1 a))) (x (- x y)))))))
```

Also formalised is a proof of each of the statements in Thm. 1. Here we outline the proof of the convexity of $a \cdot f$ given that $a \geq 0$ and f is convex. The rest are similar. Moreover, the approach we take resembles our approach to formalising Thm. 2 albeit much simpler. In particular, to reason about functions, we use the technique of encapsulation to first prove the desired theorem for a witness function. Functional instantiation then provides a method for reasoning about functions in general. Our witness, `cvfn-1`, is a constant function and so is clearly convex. This can be seen in Prog. 3.4.

Explicitly, $a \cdot f$ is convex because

$$af(\alpha x + (1-\alpha)y) \leq a(\alpha f(x) + (1-\alpha)f(y)) = \alpha(af(x)) + (1-\alpha)(af(y)). \qquad (18)$$

In particular, we invoke convexity and need to distribute a. Moreover, this line of reasoning is not dependent on the definition of f so we may disable the definition of `cvfn-1` in Prog. 3.4. This can be seen in Prog. 3.5.

Program 3.4 Encapsulating a constant function `cvnf-1` and stating its convexity.

```
(encapsulate
 (((cvfn-1 *) => *)...)

 (local (defun cvfn-1 (x) (declare (ignore x)) 1337))
 ...
 (defthm cvfn-1-convex
  (implies (and (real-listp x) (real-listp y) (= (len y) (len x))
                (realp a) (<= 0 a) (<= a 1))
           (<= (cvfn-1 (vec-+ (scalar-* a x) (scalar-* (- 1 a) y)))
               (+ (* a (cvfn-1 x)) (* (- 1 a) (cvfn-1 y)))))) ...)
```

Program 3.5 Supressing the definition of `cvfn-1`, stating a special case of distributivity, and stating the convexity of $a \cdot f$.

```
(local (in-theory (disable (:d cvfn-1) (:e cvfn-1) (:t cvfn-1))))

(encapsulate ...
  ;;  factor out alpha
  (local (defthm lemma-1
   (implies (and (real-listp x) (real-listp y) (= (len y) (len x))
                 (realp a) (<= 0 a) (<= a 1)
                 (realp alpha) (<= 0 alpha))
            (= (+ (* a (* alpha (cvfn-1 x)))
                  (* (- 1 a) (* alpha (cvfn-1 y))))
               (* alpha
                  (+ (* a (cvfn-1 x))
                     (* (- 1 a) (cvfn-1 y))))))))

  (defthm a-*-cvfn-1-convex
   (implies (and (real-listp x) (real-listp y) (= (len y) (len x))
                 (realp a) (<= 0 a) (<= a 1)
                 (realp alpha) (<= 0 alpha))
            (<= (* alpha (cvfn-1 (vec-+ (scalar-* a x) (scalar-* (- 1 a) y))))
                (+ (* a (* alpha (cvfn-1 x)))
                   (* (- 1 a) (* alpha (cvfn-1 y))))))
   :hints (("GOAL" :in-theory (disable distributivity)
                   :use ((:instance cvfn-1-convex)
                         (:instance lemma-1))))))
```

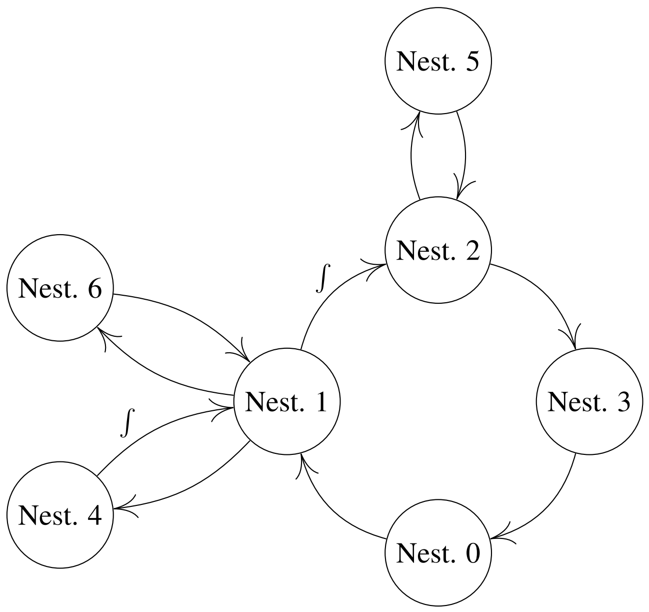


Figure 1: Nesterov's proof of Thm. 2. Here Nest. 0 is Lipschitz continuity. Integration is denoted by $\int$.

We omit the formal proofs for the other claims of Thm. 1 as well as the proofs of convexity for the Euclidean norm $\|\cdot\|_2$ and its square $\|\cdot\|_2^2$ and the inequality [1]

$$\alpha(1-\alpha)\|x-y\|^2 \leq \alpha\|x\|^2 + (1-\alpha)\|y\|^2. \tag{19}$$

4 Nesterov's Theorem in ACL2(r)

4.1 Approach

In [11], Nesterov provided a proof that followed the structure visualised by Fig. 1. Nesterov's proof, however, uses techniques that are not amenable to proofs in ACL2(r). In particular, integration is used multiple times to show some inequalities. However, integration in ACL2(r) is dependent on the function that is being integrated [7]. This places extra obligations on the user. The alternate approach shown in Fig. 2 requires fewer instances of integration than Fig. 1. Moreover, Fig. 2 has fewer implications to prove in general. The primary difference in our approach is that we prove Nest. 4 from a straightforward application of Cauchy-Schwarz and omit Nest. 1 implies Nest. 4.

Stating a theorem about functions in ACL2 is an unnatural endeavour because ACL2 is a theorem prover for first-order logic so we cannot predicate over sets in general. The natural solution is to leverage encapsulation and functional instantiation to obtain pseudo-higher order behaviour. However, this means that the desired function for which the user wishes to apply Thm. 2 must pass the theorems within the encapsulation. To formalise the theorem in its greatest generality, it is necessary to suppress the definition of the witness function in the encapsulation and instead prove the theorems based on the properties of the function. To use functional instantiation, these properties must be proven for the desired function. Thus, we aim to minimize the number of properties that the user must show, and derive as much as possible

[1] These proofs can be found in `convex.lisp`.

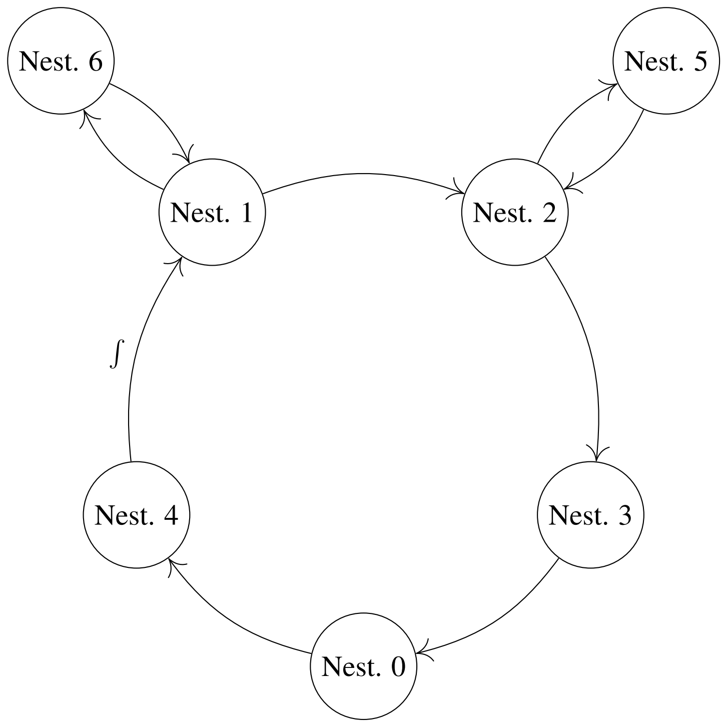


Figure 2: Another proof of Thm. 2. Here Nest. 0 is Lipschitz continuity. Integration is denoted by $\int$.

within the encapsulation. In our case, the user obligations are the theorems and identities involving the continuity, derivative, and integral of the encapsulated functions as well as any forms explicitly involving the dimension of the space.

The encapsulated functions include the multivariate function of interest `mvfn` and its derivative `nabla-mvfn`, the function that evaluates to the Lipschitz constant `L`, a helper function `phi` based on `mvfn` and its derivative `nabla-phi`, the recognizer for vectors with standard real entries `standard-vecp`, and the function, `DIM`, that evaluates to the dimension n of the vector space.

The fundamental challenges of formalising $\mathbb{R}^n$ are explored in [8]. In the context of this paper, however, the particular difficulties to note are the recognizer for vectors with standard entries, exhibiting the relationship between vectors with infinitesimal entries and the norm, and invoking two copies of certain inequalities. More details will be discussed as we proceed.

4.2 Basic Definitions & Lemmas

As an example of the formalisation, the definition of continuity for multivariate functions can be seen in Prog. 4.1. The hypotheses ensure the entries are both real vectors of the same dimension and `i-small` is the recognizer for infinitesimals. This form is consistent with the non-standard analysis definition of continuity.

The prohibition of non-classical recursive functions and the necessity of a recognizer for vectors with standard entries forces the recognizer to be defined with a specific n within the encapsulation. An encapsulation requires nevertheless a witness for the convex function of interest (which in this case happens to be the function $f(x) = 42$) so we set $n = 2$. The recognizer `standard-vecp` simply checks

Program 4.1 Continuity for a multivariate function `mvfn`.

```
(defthm mvfn-is-continuous-3
 (implies (and (real-listp vec1) (real-listp vec2)
               (= (len vec2) (len vec1)) (= (len vec1) (DIM))
               (i-small (eu-metric vec1 vec2)))
          (i-small (abs (- (mvfn vec1) (mvfn vec2))))))
```

whether a vector of dimension 2 has standards in both its coordinates.[2]

Another recurring and particularly useful theorem is the Cauchy-Schwarz inequality. It was formalised in [8] and has applications in proving the triangle inequality and the properties of $\mathbb{R}^n$ as a metric space. We use a specific version of the inequality that can be seen in Prog. 4.2.

Program 4.2 A version of the Cauchy-Schwarz inequality used in our proof of Thm. 2.

```
(defthm cauchy-schwarz-2
 (implies (and (real-listp u) (real-listp v) (= (len u) (len v)))
          (<= (abs (dot u v))
              (* (eu-norm u) (eu-norm v))))...)
```

There are several proofs of implications in Fig. 2 that follow almost immediately from the mentioned definitions and lemmas. We outline one of them: Nest. 0 implies Nest. 4. The proof mimics the chain of inequalities

$$L\|x-y\|^2 \geq \|f'(x)-f'(y)\| \cdot \|x-y\| \geq \langle f'(x)-f'(y), x-y\rangle \tag{20}$$

where the first inequality follows from Lipschitz continuity and the second inequality follows from Cauchy-Schwarz. We begin with the first inequality of Eq. 20 in Prog. 4.3. Applying the Cauchy-

Program 4.3 The first inequality follows from Lipschitz continuity.

```
;; ||f'(x) - f'(y)|| <= L||x - y|| implies
;; ||f'(x) - f'(y)|| ||x - y|| <= L||x - y||^2
(local (defthm lemma-2
 (implies (and (real-listp x) (real-listp y)
               (= (len y) (len x)) (= (len x) (DIM))
               (<= (eu-norm (vec-- (nabla-mvfn x) (nabla-mvfn y)))
                   (* (L) (eu-norm (vec-- x y)))))
          (<= (* (eu-norm (vec-- (nabla-mvfn x) (nabla-mvfn y)))
                 (eu-norm (vec-- x y)))
              (* (L) (eu-norm (vec-- x y))
                     (eu-norm (vec-- x y)))))...))
```

Schwarz inequality from Prog. 4.2 gives the second inequality of Eq. 20 under absolute values as seen in Prog. 4.4. Eliminating absolute values gives the desired inequality (in an "expanded" form) as seen in Prog. 4.5. To state the implication in its full generality and for reasons to appear in the next section, we

[2]The rest of the basic definitions and theorems can be found in `nesterov-1.lisp`.

Program 4.4 The second inequality follows from Cauchy-Schwarz.

```
;; |<f'(x) - f'(y), x - y>| <= L||x - y||^2
(local (defthm lemma-3
 (implies (and (real-listp x) (real-listp y)
               (= (len y) (len x)) (= (len x) (DIM))
               (<= (eu-norm (vec-- (nabla-mvfn x) (nabla-mvfn y)))
                   (* (L) (eu-norm (vec-- x y)))))
         (<= (abs (dot (vec-- (nabla-mvfn x) (nabla-mvfn y))
                       (vec-- x y)))
             (* (L) (eu-norm (vec-- x y))
                    (eu-norm (vec-- x y)))))
 :hints (("GOAL" :use ((:instance lemma-2)
                       ...
                       (:instance cauchy-schwarz-2
                                  (u (vec-- (nabla-mvfn x)
                                            (nabla-mvfn y)))
                                  (v (vec-- x y))))))))
```

Program 4.5 The desired implication in an expanded form.

```
;; <f'(x) - f'(y), x - y> <= L ||x - y||^2
(defthm ineq-0-implies-ineq-4-expanded
 (implies (and (real-listp x) (real-listp y)
               (= (len y) (len x)) (= (len x) (DIM))
               (<= (eu-metric (nabla-mvfn x) (nabla-mvfn y))
                   (* (L) (eu-metric x y))))
         (<= (dot (vec-- (nabla-mvfn x) (nabla-mvfn y)) (vec-- x y))
             (* (L) (metric^2 x y))))
 :hints (("GOAL" :use (...(:instance lemma-3)))))
```

use Skolem functions to replace the inequalities in the theorem. The function definitions can be seen in Prog. 4.6. The theorem then becomes of the form seen in Prog. 4.7 where the hypotheses ensure that we are dealing with real vectors. All the implications, whether they follow straight from the definitions or otherwise, are of this form.[3]

The other implications that follow mainly from the definitions are Nest. 4 implies Nest. 1 and Nest. 1 implies Nest. 2.

Program 4.6 Skolem function definitions that allow us to invoke the `forall` quantifier.

```
;; Lipschitz continuity ||f'(x) - f'(y)|| <= L ||x - y||
(defun-sk ineq-0 (L)
 (forall (x y)
  (<= (eu-metric (nabla-mvfn x) (nabla-mvfn y))
      (* L (eu-metric x y))))...)
...
;; <f'(x) - f'(y), x - y> <= L ||x - y||^2
(defun-sk ineq-4 (L)
 (forall (x y)
  (<= (dot (vec-- (nabla-mvfn x) (nabla-mvfn y)) (vec-- x y))
      (* L (metric^2 x y))))...)
```

Program 4.7 The desired implication.

```
(defthm ineq-0-implies-ineq-4
  (implies (and (hypotheses (ineq-4-witness (L)) (DIM))
                (ineq-0 (L)))
           (ineq-4 (L)))...)
```

4.3 Challenging Issues

Here we outline some of the challenges we encountered during our formalisation of Thm. 2. Several of these issues involve the proofs of the remaining lemmas, which all require some user intervention beyond simple algebraic manipulation. Here we discuss two such instances. The others are omitted because we solve them similarly. Finally, we discuss the final form of Thm. 2 and the various considerations regarding it and alternative approaches.

4.3.1 Instantiating Inequalities

The proof of Nest. 2 implies Nest. 3 amounts to adding two copies of Nest. 2 with x, y swapped. This induces issues regarding the proof of the implication. The natural form of the lemma would involve Nest. 2 among the hypotheses as in Prog. 4.8.

However, to instantiate a copy of Nest. 2 with swapped x, y in such a form would be equivalent to

$$\forall x,y, (P(x,y) \implies P(y,x)) \tag{21}$$

[3]The rest of the Skolem functions can be found in `nesterov-4.lisp`

Program 4.8 An “obvious” way to state Nest. 2 implies Nest. 3.

```
(defthm ineq-2-implies-ineq-3
  (implies (and (real-listp x) (real-listp y)
                (= (len y) (len x)) (= (len x) (DIM))
                (ineq-2 (L)))
           (ineq-3 (L)))...)
```

where P is a predicate (in this case equivalent to Nest. 2), which is not necessarily true. The form we wish to have is

$$(\forall x, y, P(x,y)) \implies (\forall x, y, P(y,x)). \tag{22}$$

In order to instantiate another copy of Nest. 2 within the implication requires quantifiers within the theorem statement. The usual approach involves using Skolem functions to introduce quantified variables. We can now instantiate the two copies with swapped variables as in Prog. 4.9.

Program 4.9 Instantiating two copies of Nest. 2 with swapped variables.

```
(defthm ineq-2-expanded-v1
  (implies (ineq-2 (L))
           (and (<= (+ (mvfn x)
                       (dot (nabla-mvfn x) (vec-- y x))
                       (* (/ (* 2 (L)))
                          (metric^2 (nabla-mvfn x) (nabla-mvfn y))))
                    (mvfn y))
                (<= (+ (mvfn y)
                       (dot (nabla-mvfn y) (vec-- x y))
                       (* (/ (* 2 (L)))
                          (metric^2 (nabla-mvfn y) (nabla-mvfn x))))
                    (mvfn x))))...)
```

We also considered simply including two copies of the inequality with swapped variables among the hypotheses. This has two advantages. Firstly, with such a form, the lemma becomes stronger because the hypothesis $P(x,y) \wedge P(y,x)$ is weaker than $\forall x, y, P(x,y)$. Secondly, the lemma is slightly easier to pass in ACL2(r). However, the primary drawback is that this form is inconsistent with the other lemmas and the final form of Nesterov’s theorem becomes less elegant (eg. showing Nest. 1 implies Nest. 2 would also require two copies of Nest. 1).

The lemmas Nest. 1 implies Nest. 6 and Nest. 2 implies Nest. 5 also requires instantiating multiple copies of the antecedent inequalities (albeit with different vectors).

4.3.2 Taking Limits

In the language of non-standard analysis, limits amount to taking standard-parts. For example, $\lim_{x \to a} f(x)$ is equivalent to $\mathrm{st}(f(x))$ when $x - a$ is an infinitesimal. However, for products, say, xy, the identity $\mathrm{st}(xy) = \mathrm{st}(x)\,\mathrm{st}(y)$ only holds when x, y are both finite reals. In the proof of Nest. 6 implies Nest. 1, there is a step that requires taking the limit of $(1-\alpha)\|y-x\|^2$ as $\alpha \to 0$. Now, if $\alpha > 0$ is an infinitesimal,

$$\mathrm{st}((1-\alpha)\|y-x\|^2) = \mathrm{st}(1-\alpha)\,\mathrm{st}(\|y-x\|^2) = \|y-x\|^2 \tag{23}$$

is easy to satisfy when x, y are vectors with standard real components. Moreover, requiring variables to be standard is consistent with some instances of single variable theorems (eg. the product of continuous functions is continuous). It then remains to state such a hypothesis using, say, a recognizer `standard-vecp` as in Prog. 4.10. The natural approach to defining `standard-vecp` would be to simply

Program 4.10 Introducing `standard-vecp` into the hypotheses.

```
(defthm ineq-6-implies-ineq-1-expanded
 (implies (and (real-listp x) (real-listp y)
              (= (len y) (len x)) (= (len x) (DIM))
              (realp alpha) (i-small alpha)
              (< 0 alpha) (<= alpha 1)
              (standard-vecp x) (standard-vecp y)
              (<= (+ (* alpha (mvfn y))
                     (* (- 1 alpha) (mvfn x)))
                  (+ (mvfn (vec-+ (scalar-* alpha y)
                                  (scalar-* (- 1 alpha) x)))
                     (* (/ (L) 2) alpha (- 1 alpha) (metric^2 y x)))))
         (<= (mvfn y)
             (+ (mvfn x)
                (dot (nabla-mvfn x) (vec-- y x))
                (* (/ (L) 2) (metric^2 y x))))))...)
```

recurse on the length of a vector applying `standardp` to each entry. However, `standardp` is non-classical and this definition encounters a common issue throughout our ACL2(r) formalisation in that it is a non-classical recursive function. We discuss the subtleties of this problem more in [8]. Because `standard-vecp` is dependent on the dimension, our solution is to encapsulate the function and prove the necessary theorems involving it (eg. `metric^2` is `standardp` on `standard-vecp` values). For the case $n = 2$, we simply check the length of the vector is two and that both entries are standard reals. A final note regarding the lemma is the hypothesis $\alpha > 0$ replacing the weaker $\alpha \geq 0$ since part of the proof depends on dividing by α.

The other lemma that requires similar hypotheses is the implication Nest. 5 implies Nest. 2.

4.3.3 Final Form of Nesterov's Theorem

Finally, we discuss our final form of Thm. 2 as well as several alternatives and their considerations. The final form can be seen in Prog. 4.11. The function `hypotheses` ensure that we are dealing with real vectors of the correct dimension. The function `st-hypotheses` ensure that the vectors have standard entries due to the necessity of taking limits. The function `alpha-hypotheses` is the same as `hypotheses` but includes the hypothesis that $\alpha \in [0,1]$. The function `alpha->-0-hypotheses` ensures that $\alpha > 0$ for the case of taking limits.

In Sec. 4.2, we already mentioned, for each lemma, the basic structure involving Skolem functions. In Sec. 4.3.1, we cited elegance and ease of instantiation as reasons for using Skolem functions. Because stating even the shorter inequalities in Polish notation would quickly become awkward and unintelligible (eg. Prog. 4.12), it became desirable for us to define the inequalities in a clean and clear manner. One simple approach would be to define the inequalities as ACL2 functions or macros. Unfortunately, during the course of our formalisation, we found that the rewriter would be tempted to "simplify" or otherwise

Program 4.11 The final statement of Thm. 2.

```
(defthm nesterov
 ;; theorem statement
 (implies (and (hypotheses (ineq-0-witness (L)) (DIM))
                   (hypotheses (ineq-1-witness (L)) (DIM))
                   (hypotheses (ineq-2-witness (L)) (DIM))
                   (hypotheses (ineq-3-witness (L)) (DIM))
                   (hypotheses (ineq-4-witness (L)) (DIM))
                   (st-hypotheses (ineq-1-witness (L)))
                   (st-hypotheses (ineq-2-witness (L)))
                   (alpha-hypotheses (ineq-5-witness (L)) (DIM))
                   (alpha-hypotheses (ineq-6-witness (L)) (DIM))
                   (alpha->-0-hypotheses (ineq-5 (L)))
                   (alpha->-0-hypotheses (ineq-6 (L)))
                   (or (ineq-0 (L)) (ineq-1 (L)) (ineq-2 (L)) (ineq-3 (L))
                       (ineq-4 (L)) (ineq-5 (L)) (ineq-6 (L))))
          (and (ineq-0 (L)) (ineq-1 (L)) (ineq-2 (L)) (ineq-3 (L))
               (ineq-4 (L)) (ineq-5 (L)) (ineq-6 (L))))
 ;; hints, etc. for ACL2(r)
 ...)
```

change the form of the inequality via arithmetic rules. This made applications of certain theorems more involved and arduous than necessary. Therefore, it would be necessary to disable the function definitions anyways. In addition to permitting instantiations of inequalities with different vectors within a single theorem statement, Skolem functions would allow us to suppress or "hide" the explicit inequality thus providing a clear, concise, and compact package.

Program 4.12 Nest. 5 in Polish notation.

```
(<= (+ (mvfn (vec-+ (scalar-* alpha y) (scalar-* (- 1 alpha) x)))
       (* (/ (* 2 (L))) (* alpha (- 1 alpha) (metric^2 (nabla-mvfn y)
                                                      (nabla-mvfn x)))))
    (+ (* alpha (mvfn y)) (* (- 1 alpha) (mvfn x))))
```

On the other hand, this form has the unfortunate drawback of making the proof of the theorem slightly more involved. By introducing Skolem functions we also introduce the necessity of witness functions; proving the lemmas in terms of the witness functions may occasionally become onerous. For example, to state the hypotheses that the entries of the witness functions are real vectors of the same dimension, we would like to define a `hypotheses` function. However, explicitly exhibiting the witness functions within the definition of `hypotheses` leads to a signature mismatch. We instead pass the witness function to `hypotheses` as an argument.

5 Conclusion

In this paper, we presented a set of theorems for reasoning about convex functions in ACL2(r). We also discussed some of the challenges of formalising a proof that relies heavily on informally well-established and intuitive notions. Examples include translating statements in classical multivariate analysis into non-standard analysis and instantiating quantified statements using Skolem functions. Our particular interest in this work are the potential applications to verifying, among other areas, optimization algorithms used in machine learning. To this end, we chose a theorem of Nesterov's to serve as an example of the analytical reasoning possible in our formalisation. The natural next step would be to develop a proper theory of multivariate calculus to further automate the reasoning about optimisation algorithms.

References

[1] Leif O. Arkeryd, Nigel J. Cutland & C. Ward Henson (Eds.) (1997): *Nonstandard Analysis: Theory and Applications*, 1st edition. *Nato Science Series C: 493* , Springer Netherlands, doi:10.1007/978-94-011-5544-1.

[2] Stephen Boyd & Lieven Vandenberghe (2004): *Convex Optimization.* Cambridge University Press, doi:10.1017/CBO9780511804441.

[3] Ruben Gamboa (2000): *Continuity and Differentiability.* In Matt Kaufmann, Panagiotis Manolios & J. Strother Moore, editors: *Computer-Aided Reasoning: ACL2 Case Studies*, Springer US, Boston, MA, pp. 301–315, doi:10.1007/978-1-4757-3188-0_18.

[4] Ruben A. Gamboa & Matt Kaufmann (2001): *Nonstandard Analysis in ACL2. Journal of Automated Reasoning* 27(4), pp. 323–351, doi:10.1023/A:1011908113514.

[5] John Harrison (2007): *Formalizing Basic Complex Analysis.* In R. Matuszewski & A. Zalewska, editors: *From Insight to Proof: Festschrift in Honour of Andrzej Trybulec*, *Studies in Logic, Grammar and Rhetoric* 10(23), University of Białystok, pp. 151–165. Available at `http://mizar.org/trybulec65/`.

[6] Nathan Jacobson (1985): *Basic Algebra I*, 2nd edition. Dover Publications.

[7] Matt Kaufmann (2000): *Modular Proof: The Fundamental Theorem of Calculus.* In Matt Kaufmann, Panagiotis Manolios & J. Strother Moore, editors: *Computer-Aided Reasoning: ACL2 Case Studies*, Springer US, Boston, MA, pp. 75–91, doi:10.1007/978-1-4757-3188-0_6.

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

[9] Serge Lang (2002): *Algebra*, 3rd edition. *Graduate Texts in Mathematics 211* , Springer-Verlag New York, doi:10.1007/978-1-4613-0041-0.

[10] Peter A. Loeb & Manfred P. H. Wolff (2015): *Nonstandard Analysis for the Working Mathematician*, 2nd edition. Springer Netherlands, doi:10.1007/978-94-017-7327-0.

[11] Yurii Nesterov (2004): *Introductory Lectures on Convex Optimization*, 1st edition. *Applied Optimization 87* , Springer US, doi:10.1007/978-1-4419-8853-9.

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

[13] Steven Roman (2008): *Advanced Linear Algebra*, 3rd edition. *Graduate Texts in Mathematics 135* , Springer-Verlag New York, doi:10.1007/978-0-387-72831-5.

[14] Walter Rudin (1976): *Principles of Mathematical Analysis*, 3rd edition. *International Series in Pure and Applied Mathematics* , McGraw-Hill.

[15] Georgi E. Shilov (1977): *Linear Algebra.* Dover Publications.

Smtlink 2.0

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

`Smtlink` is an extension of ACL2 with Satisfiability Modulo Theories (SMT) solvers. We presented an earlier version at ACL2'2015. `Smtlink` 2.0 makes major improvements over the initial version with respect to soundness, extensibility, ease-of-use, and the range of types and associated theory-solvers supported. Most theorems that one would want to prove using an SMT solver must first be translated to use only the primitive operations supported by the SMT solver – this translation includes function expansion and type inference. `Smtlink` 2.0 performs this translation using a sequence of steps performed by verified clause processors and computed hints. These steps are ensured to be sound. The final transliteration from ACL2 to Z3's Python interface requires a trusted clause processor. This is a great improvement in soundness and extensibility over the original `Smtlink` which was implemented as a single, monolithic, trusted clause processor. `Smtlink` 2.0 provides support for FTY `defprod`, `deflist`, `defalist`, and `defoption` types by using Z3's arrays and user-defined data types. We have identified common usage patterns and simplified the configuration and hint information needed to use `Smtlink`.

1 Introduction

Interactive theorem proving and SMT solving are complementary verification techniques. SMT solvers can automatically discharge proof goals with thousands to hundreds of thousands of variables when the goals are within the theories of the solver. These theories include linear and non-linear arithmetic, uninterpreted functions, arrays, and bit-vector theories. On the other hand, verification of realistic hardware and software systems often involves models with a rich variety of data structures. Their proofs involve induction, encapsulation (or other forms of higher-order reasoning), and careful design of rules to avoid pushing the solver over an exponential cliff of search complexity. Ideally, we would like to use the capabilities of SMT solvers to avoid proving large numbers of simple but tedious lemmas and combine those results in an interactive theorem prover to enable the reasoning of properties in large, realistic hardware and software designs.

`Smtlink` is a book developed for integrating SMT solvers into ACL2. In the previous work [22], we implemented an interface using a large trusted clause-processor, and only supported a limited subset of SMT theories. In this work, we introduce an architecture based on a collection of verified clause processors that transform an ACL2 goal into a form amenable for discharging with an SMT solver. The final step transliterates the ACL2 goal into the syntax of the SMT solver, invokes the solver, and discharges the goal or reports a counter-example based on the results from the solver. This final step is, performed by a trusted-clause processor because we are trusting the SMT solver. Because most of the translation is performed by verified clause processors, the soundness for those steps is guaranteed. Furthermore, the task of the final, trusted clause processor is simple to describe, and that description suggests the form for the corresponding soundness argument. The modularized architecture is readily extensible which makes introducing new clause transformation steps straightforward.

*This work is supported in part by the Institute for Computing, Information and Cognitive Systems (ICICS) at UBC.

Our initial motivation for linking SMT solvers with ACL2 was for proving linear and non-linear arithmetic for Analog and Mixed-Signal (AMS) designs [23]. This original version supported the SMT solver's theories of boolean satisfiability and integer and rational arithmetic, but provided no support for other types. For example, to verify theorems involving lists, one would need to expand functions involving lists to a user-specified depth, treat remaining calls as uninterpreted functions which must return a boolean, integer, rational, or real. The lack of support for a rich set of types restricted the applicability of the original `Smtlink` to low-level lemmas that are primarily based on arithmetic. The new `Smtlink` supports symbols, lists, and algebraic datatypes, with a convenient interface to FTY types.

Other changes include better support for function expansion and uninterpreted functions. The new `smtlink-hint` interface is simpler. `Smtlink` generates auxiliary subgoals for ACL2 in "obvious" cases, such as when the user adds a hypothesis for `Smtlink` that is not stated in the original goal. Hints for these subgoals share the same structure as the ACL2 hints and are attached by the `Smtlink` hint syntax to the relevant subgoal – no subgoal specifiers are required! When the SMT solver refutes a goal, the putative counterexample is returned to ACL2 for user examination. Currently, `Smtlink` supports the Z3 SMT solver [20] using Z3's Python API. `Smtlink` is compatible with both Python 2 and Python 3, and can be used in ACL2(r) with `realp`.

Documentation is online under the topic `:doc smtlink`. It describes how to install Z3, configure `Smtlink`, certify `Smtlink` and test the installation. It also describes the new interface of the `Smtlink` hint, contains several tutorial examples, and some developer documentation. Examples shown in this paper assumes proper installation and setups are done as is described in `:doc smtlink`. It is our belief that given a more compelling argument of soundness for the architecture, a richer set of supported types or theories, and a more user-friendly user interface, `Smtlink` can support broader and larger proofs.

In Section 2 we present the architecture of `Smtlink` based on verified clause processors, hint wrappers, and computed hints. Section 3 describes the SMT theories supported by `Smtlink` and sketches the soundness argument. Section 4 gives a simple example where we use lists, alists, symbols, and booleans to model the behavior of a ring oscillator circuit. Section 5 provides a summary of related work, and we summarize the paper along with describing ongoing and planned extensions to `Smtlink` in Section 6.

2 Smtlink 2.0 Architecture

This section describes the architecture of `Smtlink` 2.0. We first introduce a running example to illustrate each of the steps taken by the clause processor. Section 2.2 describes the top-level architecture, followed by a brief description of each of the clause processors used in `Smtlink`.

2.1 An Nonlinear Inequality Example

To illustrate the architecture of `Smtlink`, we use a running example throughout this section. Consider proving the following theorem in ACL2:

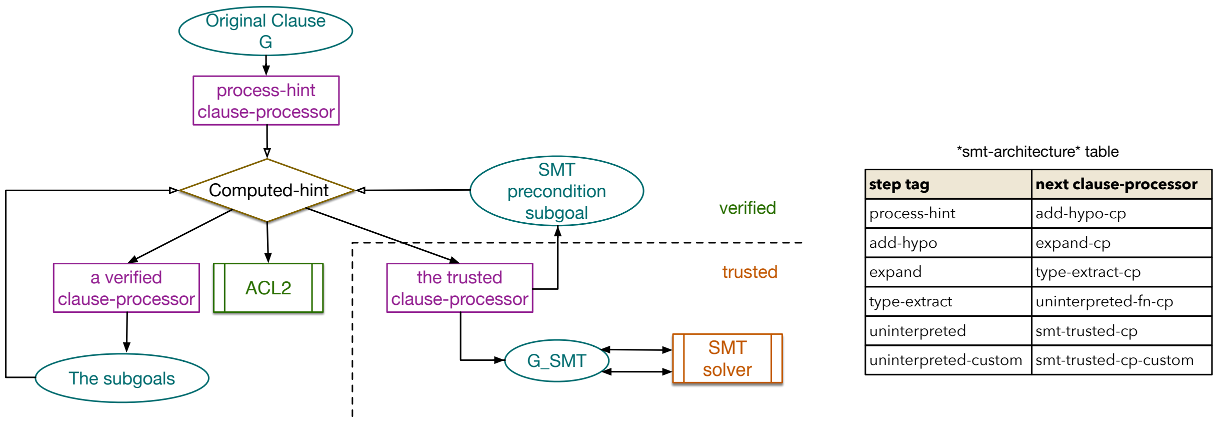
Theorem 2.1 *$\forall x \in R$ and $\forall y \in R$, if $\frac{9x^2}{8} + y^2 \leq 1$ and $x^2 - y^2 \leq 1$, then $y < 3(x - \frac{17}{8})^2 - 3$.*

Program 2.1 shows the corresponding theorem definition in ACL2. To use `Smtlink` to prove a theorem, a hint `:smtlink [smt-hint]` can be provided using ACL2's standard `:hints` interface. `SMT::smt-hint` is the hint provided to `Smtlink`. `SMT::smt-hint` follows a structure that is described in `:doc smt-hint`. In this example, `:smtlink nil` suggests using `Smtlink` without any additional hints provided to `Smtlink`. The initial goal (underlining represented as clauses) is:

Program 2.1 A nonlinear inequality problem

```
(defun x^2-y^2 (x y) (- (* x x) (* y y)))

(defthm poly-ineq-example
  (implies (and (real/rationalp x) (real/rationalp y)
                (<= (+ (* (/ 9 8) x x) (* y y)) 1)
                (<=  (x^2-y^2 x y) 1))
           (< y (- (* 3 (- x (/ 17 8)) (- x (/ 17 8))) 3)))
  :hints(("Goal"
          :smtlink nil)))
```



step tag	next clause-processor
process-hint	add-hypo-cp
add-hypo	expand-cp
expand	type-extract-cp
type-extract	uninterpreted-fn-cp
uninterpreted	smt-trusted-cp
uninterpreted-custom	smt-trusted-cp-custom

Figure 1: Smtlink Architecture

```
(IMPLIES (AND (RATIONALP X) (RATIONALP Y)
              (<= (+ (* (* 9 (/ 8)) X X) (* Y Y)) 1)
              (<= (X^2-Y^2 X Y) 1))
         (< Y (+ -3 (* 3 (+ X (- (* 17 (/ 8))))
                         (+ X (- (* 17 (/ 8))))))))
```

2.2 The Architecture

Let G denote the goal to be proven. As shown in Figure 1, the `Smtlink` hint invokes the verified clause processor called `SMT::process-hint`. The arguments to this clause-processor are the clause, G, and a list of hints provided by the user. The `SMT::process-hint` performs syntactic checks on the user's hints and translates them into an internal representation used by the subsequent clause processors. The user can specify default hints to be used with all invocations of `Smtlink`. These are merged with any hints that are specific for this theorem to produce `combined-hint`. The `SMT::process-hint` adds a `SMT::hint-please` wrapper to the clause and returns a term of the form

```
`(((SMT::hint-please (:clause-processor (SMT::add-hypo-cp clause ,combined-hint))) ,@G))
```

`Smtlink` 2.0 uses the hint wrapper approach from `books/hints/hint-wrapper.lisp`. In particular, we define a "hint-wrapper" called `SMT::hint-please` in package `smtlink` that always returns `nil`.

Clauses in ACL2 are represented as lists of disjuncts, and our hint-wrapper can be added as a disjunct to any clause without changing the validity of the clause. `SMT::hint-please` takes one input argument – the list of hints.

The computed-hint called `SMT::SMT-computed-hint` searches for a disjunct of the form `(SMT::hint-please ...)` in each goal generated by ACL2. When it finds such an instance, the `SMT::SMT-computed-hint` will return a `computed-hint-replacement` of the form:

```
‘(:computed-hint-replacement
  ((SMT::SMT-computed-hint clause))
  (:clause-processor (SMT::some-verified-cp clause ,combined-hint)))
```

This applies the next verified clause-processor called `SMT::some-verified-cp` to the current subgoal and installs the computed-hint `SMT::SMT-computed-hint` on subsequent subgoals again. The `SMT::some-verified-cp` clause-processor is one step in a sequence of verified clause processors. `Smtlink` uses a configuration table called `*smt-architecture*` to specify the sequence of clause processors, as shown in Figure 1. Each clause processor consults this table to determine its successor. By updating this table, `Smtlink` is easily reconfigured.

As described above, the initial clause processor for `Smtlink`, `SMT::process-hint`, adds a `SMT::hint-please` disjunct to the clause to indicate that the clause processor, `SMT::add-hypo-cp` should be the next step in translating the clause to a form amenable for a SMT solver.

2.2.1 add-hypo-cp

A key to using an SMT solver effectively is to tell it what it needs to know but to avoid overwhelming it with facts that push it over an exponential complexity cliff. The user can guide this process by adding hypotheses for the SMT solver – typically these are facts from the ACL2 logical world that don't need to be stated as hypotheses of the theorem. The clause-processor `SMT::add-hypo-cp` is a verified clause processor that adds user-provided hypotheses to the goal. Let **G** denote the original goal and $\mathbf{H_1}$, ... $\mathbf{H_n}$ denote the new hypotheses. Each added hypothesis is returned by the clause processors as a subgoal to be discharged by ACL2. The user can attach hints to these hypotheses – for example, showing that the hypothesis is a particular instantiation of a previously proven theorem. The soundness of `SMT::add-hypo-cp` is established by the theorem:

$$\frac{H_1 \vee G \quad \ldots \quad H_n \vee G \quad \bigwedge_{i=1}^{N} H_i \Rightarrow G}{G} \tag{1}$$

The term $\bigwedge_{i=1}^{N} H_i \Rightarrow G$ is the main clause that gets passed onto the next clause-processor. In the following, let G_{hyp} denote $\bigwedge_{i=1}^{N} H_i \Rightarrow G$.

As for the example in Program 2.1, the user didn't provide any additional guidance. We see that the clauses generated are almost unchanged. The only place that changed, is that `Smtlink` has installed the next clause-processor to be `SMT::expand-cp`.

```
(IMPLIES (NOT (SMT::HINT-PLEASE '(:CLAUSE-PROCESSOR (SMT::EXPAND-CP CLAUSE ...))))
 (IMPLIES (AND (RATIONALP X) (RATIONALP Y)
               (<= (+ (* (* 9 (/ 8)) X X) (* Y Y)) 1)
               (<= (X^2-Y^2 X Y) 1))
          (< Y (+ -3 (* 3 (+ X (- (* 17 (/ 8))))
                      (+ X (- (* 17 (/ 8)))))))))
```

2.2.2 expand-cp

The clause-processor `SMT::expand-cp` expands function definitions to produce a goal where all operations have SMT equivalents. Function definitions are accessed using `meta-extract-formula`. By default, all non-recursive functions including disabled functions are expanded. Recursive functions are expanded once and then treated as uninterpreted functions. We attempt one level of expansion for recursive functions given the reasoning that if the theorem can indeed be proved only knowing the return type of the function, it should still be provable when expanded once. If proving the theorem requires expansion of more than one level, or if all occurrences of the function should uninterpreted, then we rely on the user to specify this in a hint to `Smtlink`. Let $\mathbf{G}_{\text{hyp}}$ be the clause given to `SMT::expand-cp` and $\mathbf{G}_{\text{expand}}$ be the expanded version. The soundness of `SMT::expand-cp` is established by the theorem:

$$\frac{G_{\text{expand}} \Rightarrow G_{\text{hyp}} \quad G_{\text{expand}}}{G_{\text{hyp}}} \tag{2}$$

Presently, the clause $G_{\text{expand}} \Rightarrow G_{\text{hyp}}$ is returned to ACL2 for proof, and G_{expand} is the main clause that gets passed onto the next clause-processor.

For the running example, function expansion produces the two subgoals. The main clause that get passed onto the next clause-processor is:

```
(IMPLIES (NOT (SMT::HINT-PLEASE '(:CLAUSE-PROCESSOR (SMT::TYPE-EXTRACT-CP CLAUSE ...))))
 (IMPLIES (AND (RATIONALP X) (RATIONALP Y)
              (<= (+ (* (* 9 (/ 8)) X X) (* Y Y)) 1)
              (<= (LET NIL (+ (* X X) (- (* Y Y)))) 1))
          (< Y (+ -3 (* 3 (+ X (- (* 17 (/ 8))))
                        (+ X (- (* 17 (/ 8)))))))))
```

It tells `Smtlink` the next clause-processor is `SMT::type-extract-cp` which does type declaration extraction. The other subgoal is:

```
(IMPLIES
 (AND (NOT (SMT::HINT-PLEASE '(:IN-THEORY (ENABLE X^2-Y^2))))
      (IMPLIES (AND (RATIONALP X) (RATIONALP Y)
                    (<= (+ (* (* 9 (/ 8)) X X) (* Y Y)) 1)
                    (<= (LET NIL (+ (* X X) (- (* Y Y)))) 1))
               (< Y (+ -3 (* 3 (+ X (- (* 17 (/ 8))))
                              (+ X (- (* 17 (/ 8))))))))))
 (OR ... ;; some term essentially equal to nil
     (IMPLIES (AND (RATIONALP X) (RATIONALP Y)
                   (<= (+ (* (* 9 (/ 8)) X X) (* Y Y)) 1)
                   (<= (X^2-Y^2 X Y) 1))
          (< Y (+ -3 (* 3 (+ X (- (* 17 (/ 8))))
                        (+ X (- (* 17 (/ 8))))))))))
```

This second subgoal essentially proves that the expanded clause implies the original clause. Notice how `SMT::hint-please` is used again for passing other sorts of hints (that are not clause-processor hints) to ACL2 for help with the proof.

2.2.3 type-extract-cp

The logic of ACL2 is untyped; whereas SMT solvers such as Z3 use a many-sorted logic. To bridge this gap, the clause-processor `SMT::type-extract-cp` is a verified clause processor for extracting type information for free variables from the hypotheses of a clause. `SMT::type-extract-cp` traverses the clause and identifies terms that syntactically are hypotheses of the form `(type-p var)` where `type-p` is a known type recognizer, and `var` is a symbol. Let G_{expand} denote the clause given to `SMT::type-extract-cp`; $\mathbf{T_1}$, ... $\mathbf{T_m}$ denote the extracted type hypotheses; $\mathbf{G}_{\text{expand}\backslash\text{type}}$ denote G_{expand} with the type-hypotheses removed; and

$$G_{\text{type}} = (\texttt{SMT::type-hyp (list } T_1 ... T_m\texttt{):type}) \Rightarrow G_{\text{expand}\backslash\text{type}} \tag{3}$$

where `SMT::type-hyp` logically computes the conjunction of the elements in the list (`list` T_1 ... T_m) – using `SMT::type-hyp` makes these hypotheses easily identified by subsequent clause processors. The soundness of `SMT::type-extract-cp` is established by the theorem:

$$\frac{G_{\text{type}} \Rightarrow G_{\text{expand}} \quad G_{\text{type}}}{G_{\text{expand}}} \tag{4}$$

$G_{\text{type}} \Rightarrow G_{\text{expand}}$ is the auxiliary clause returned back into ACL2 for proof. G_{type} is the main clause that gets passed onto the next clause-processor.

For the running example, the main clause that gets passed onto the next clause-processor is:

```
(IMPLIES (AND (NOT (SMT::HINT-PLEASE
                     '(:CLAUSE-PROCESSOR (SMT::UNINTERPRETED-FN-CP CLAUSE ...))))
              (SMT::TYPE-HYP (HIDE (LIST (RATIONALP X) (RATIONALP Y))) :TYPE))
 (IMPLIES (AND (<= (+ (* (* 9 (/ 8)) X X) (* Y Y)) 1)
              (<= (LET NIL (+ (* X X) (- (* Y Y)))) 1))
          (< Y (+ -3 (* 3 (+ X (- (* 17 (/ 8))))
                     (+ X (- (* 17 (/ 8)))))))))
```

The other auxiliary clause is:

```
(IMPLIES (AND (NOT (SMT::HINT-PLEASE
                     '(:IN-THEORY (ENABLE SMT::HINT-PLEASE SMT::TYPE-HYP)
                       :EXPAND ((:FREE (SMT::X) (HIDE SMT::X))))))
              (IMPLIES (SMT::TYPE-HYP (HIDE (LIST (RATIONALP X) (RATIONALP Y))) :TYPE)
                (IMPLIES (AND (<= (+ (* (* 9 (/ 8)) X X) (* Y Y)) 1)
                              (<= (LET NIL (+ (* X X) (- (* Y Y)))) 1))
                         (< Y (+ -3 (* 3 (+ X (- (* 17 (/ 8))))
                                         (+ X (- (* 17 (/ 8))))))))))
 (IMPLIES (AND (RATIONALP X) (RATIONALP Y)
              (<= (+ (* (* 9 (/ 8)) X X) (* Y Y)) 1)
              (<= (LET NIL (+ (* X X) (- (* Y Y)))) 1))
              (< Y (+ -3  (* 3 (+ X (- (* 17 (/ 8))))
                             (+ X (- (* 17 (/ 8)))))))))
```

This clause is returned for proof by ACL2. The proof is straightforward – essentially ACL2 simply needs to confirm that the terms we identified as type-hypotheses really are hypotheses of the goal.

2.2.4 uninterpreted-fn-cp

Useful facts about recursive functions can be proven using the SMT solver's support for uninterpreted functions. As with variables, the return-types for these functions must be specified. For soundness, `Smtlink` must show that the ACL2 function satisfies the user given type constraints. This is done by `SMT::uninterpreted-fn-cp`. Let $\mathbf{G}_{\text{type}}$ denote the clause given to `SMT::uninterpreted-fn-cp`, and let $\mathbf{R_1}$, ... $\mathbf{R_p}$ denote the assertions about the types for each call to an uninterpreted function. Let $\mathbf{Q_1}$ be the list of clauses

$$\begin{aligned} Q_1 \quad = \quad & (\texttt{SMT::type-hyp}\ (\texttt{list}\ R_1)\ \texttt{:return}) \vee G_{\text{type}}\ ,\ \dots\ , \\ & (\texttt{SMT::type-hyp}\ (\texttt{list}\ R_p)\ \texttt{:return}) \vee G_{\text{type}} \end{aligned} \tag{5}$$

Finally, let $\mathbf{G_{tcp}}$ be the clause

$$G_{tcp} = \bigwedge_{i=1}^{p} (\texttt{SMT::type-hyp}\ (\texttt{list}\ R_i)\ \texttt{:return}) \Rightarrow G_{\text{type}} \tag{6}$$

The soundness of `expand-cp` is established by the theorem:

$$\frac{Q_1 \quad G_{tcp}}{G_{\text{type}}} \tag{7}$$

The list of clauses Q_1 is returned to ACL2 for proof, and G_{tcp} is tagged with a hint to be checked by the trusted clause processor.

Our running example does not make use of uninterpreted functions in the SMT solver; so, the clause G_{tcp} is the same as G_{type} except for the detail that the `SMT::hint-please` term now specifies that the next clause-processor to use is the final trusted clause-processor `SMT::smt-trusted-cp`:

```
(IMPLIES (NOT (SMT::HINT-PLEASE '(:CLAUSE-PROCESSOR (SMT::SMT-TRUSTED-CP CLAUSE ... STATE))))
 (OR (NOT (SMT::TYPE-HYP (HIDE (LIST (RATIONALP X) (RATIONALP Y))) :TYPE))
     (IMPLIES (AND (<= (+ (* (* 9 (/ 8)) X X) (* Y Y)) 1)
                   (<= (LET NIL (+ (* X X) (- (* Y Y)))) 1))
              (< Y (+ -3 (* 3 (+ X (- (* 17 (/ 8))))
                                 (+ X (- (* 17 (/ 8))))))))))
```

2.2.5 smtlink-trusted-cp

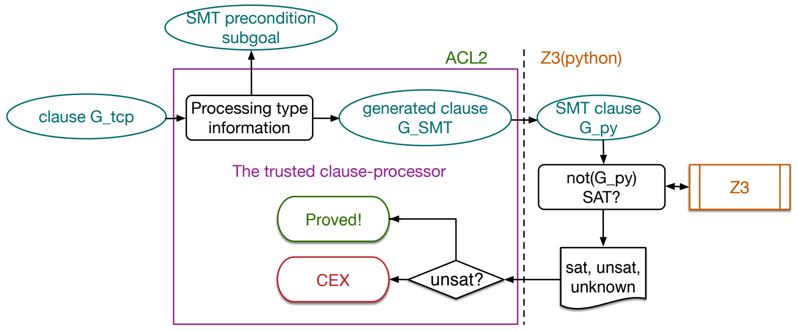


Figure 2: The trusted clause processor

Figure 2 shows the internal architecture of the trusted clause-processor, for which its correctness hasn't been proven. `Smtlink` returns counter-examples generated by the SMT solver back into ACL2. The Python interface to Z3 for `Smtlink` includes code to translate a counterexample from Z3 into list-syntax for ACL2. The form is not necessarily in ACL2 syntax. We wrote a printing function that takes the Z3 counter-examples and prints it out in an ACL2-readable form. These are all done through the trusted clause processor. Currently, the forms returned into ACL2 are not evaluable. For example, counter-examples for real numbers can take the form of an algebraic number, e.g.

```
((Y (CEX-ROOT-OBJ Y STATE (+ (^ X 2) (- 2)) 1)) (X -2))
```

We plan to generate evaluable forms in the future. Learn more about counter-example generation, see `:doc tutorial`.

The trusted clause-processor returns a subgoal called “SMT precondition” back into ACL2. By ensuring that certain preconditions are satisfied, we are able to bridge the logical meaning for tricky cases, and therefore ensure soundness. We will see explanations on this issue in Section 3. For this running example, there are no preconditions to be satisfied and the following clause trivially holds:

```
(IMPLIES (AND (NOT (SMT::HINT-PLEASE
                     '(:IN-THEORY (ENABLE SMT::MAGIC-FIX SMT::HINT-PLEASE SMT::TYPE-HYP)
                       :EXPAND ((:FREE (SMT::X) (HIDE SMT::X))))))
              (NOT (AND (OR (NOT (<= (+ (* (* 9 (/ 8)) X X) (* Y Y)) 1))
                            (LET NIL T))
                        T)))
 (OR (NOT (SMT::TYPE-HYP (HIDE (LIST (RATIONALP X) (RATIONALP Y))) :TYPE))
     (IMPLIES (AND (<= (+ (* (* 9 (/ 8)) X X) (* Y Y)) 1)
                   (<= (LET NIL (+ (* X X) (- (* Y Y)))) 1))
              (< Y (+ -3 (* 3 (+ X (- (* 17 (/ 8))))
                              (+ X (- (* 17 (/ 8))))))))))
```

2.3 A Few Notes about the Architecture

The new architecture clearly separates what’s verified from what’s trusted. As is shown in Figure 1, given an original goal, the `Smtlink` workflow goes through a series of verified clause-processor, which won’t change the logical meaning of the original goal. A computed-hint is installed to provide hints for the next step, but won’t change the logical meaning of the goals either. The final trusted clause-processor takes a goal that logically implies the original goal and derives information needed for the translation through three sources of information it uses. First, information encoded in the clause using `SMT::type-hyp`. This is sound because the definition of `SMT::type-hyp` is a conjunction of the input boolean list. Second, information about FTY types from `fty::flextypes-table`. Presently, we trust this table to be correct and intact. We plan to use rules about FTY types to derive this information in the future. Third, we use information stored in `SMT::smt-hint` about configurations, including what is the Python command, whether to treat integers as reals, where is the `ACL2_to_Z3` class file and so on. We believe this information can not accidentally introduce soundness issues from the user. For experimenters and developers, we allow this low-level interface to be overridden by the user using `:smtlink-custom` hint. Using such a “customized” `Smtlink` requires a different trust-tag than the standard version, thus providing a firewall between experiments and production versions. See `:doc tutorial` for how to use customizable `Smtlink`.

3 Types, Theories, and Soundness

`Smtlink` translates the original goal, G to an expanded goal, G_{tcp} for the trusted clause processor through a series of verified clause processors. Thus, we regard the translation to G_{tcp} as sound. The trusted clause processor translates G_{tcp} into a form that can be checked by the SMT solver; we refer to this translated form as G_{smt}. Let $x_1, x_2, \ldots, x_n$ denote the free variables in G_{tcp}, let G_{smt} denote the translated goal, and

$\tilde{x}_1, \tilde{x}_2, \ldots, \tilde{x}_n$ denote the free variables of G_{smt}. For soundness, we want

$$\text{SMT} \vdash G_{\text{smt}} \quad \Rightarrow \quad \text{ACL2} \vdash G_{\text{tcp}} \tag{8}$$

In the remainder, we assume

- ACL2 and the SMT solver are both sound for their respective theories.
- The SMT solver is a decision procedure for a decidable fragment of first-order logic. In particular, this holds for Z3, the only SMT solver that is currently supported by `Smtlink`. In addition, we are working with a quantifier-free fragment of Z3's logic.
- There is a one-to-one correspondence between the free variables of G_{tcp} and the free variables of G_{smt}. This is the case with the current implementation of `Smtlink`.

Now, suppose that G_{tcp} is not a theorem in ACL2. Then, by Gödel's Completeness Theorem, there exists a model of the ACL2 axioms that satisfies $\neg G_{\text{tcp}}$. We need to show that in this case there exists a model of the SMT solver's axioms that satisfies $\neg G_{\text{smt}}$. There are two issues that we must address. First, we need to provide, for the interpretation of any function symbol $\texttt{f}_{\text{acl2}}$ in G_{tcp}, an interpretation for the corresponding function symbol $\texttt{f}_{\text{smt}}$ in G_{smt}. This brings us to the second issue: the logic of ACL2 is untyped, but the logic of SMT solvers including Z3 is many-sorted. Thus, there are models of the ACL2 axioms that have no correspondence with the models of the SMT solver. We restrict our attention to goals, G_{tcp} where the type of each subterm in the formula can be deduced. We refer to such terms as translatable. If G_{tcp} is not translatable, then `Smtlink` will fail to prove it.

For the remainder, we restrict our attention to translatable goals. Because G_{tcp} is translatable, there is a set R of unary recognizer functions (primitives such as rationalp that return a boolean) and also a set S of other functions, such that every function symbol in G_{tcp} is a member of R or of S, and every function in S is "well-typed" with respect to R in some sense that we can define roughly as follows. We associate each function symbol $\texttt{f}_{\text{acl2}}$ in S with a function symbol $\texttt{f}_{\text{smt}}$ of Z3, and each predicate r in R with a type in Z3. The trusted clause processor checks that there is a "type-hypothesis" associated with every free variable of G_{tcp} and a fixing function for every type-ambiguous constant (e.g. `nil`) – G_{tcp} holds trivially if any of these type-hypotheses are violated. For every function $\texttt{f}_{\text{acl2}}$ in S, we associate a member of R to each of its arguments (i.e. a "type") and also to the result. For user-defined functions (i.e. uninterpreted function for the SMT solver), `Smtlink` generates a subgoal for each call to $\texttt{f}_{\text{smt}}$: if the arguments satisfy their declared types (i.e., predicates from R), then the result must satisfy its declared type as well. For built-in ACL2 functions (e.g. `binary-plus`) we assume the "obvious" theorems are present in the ACL2 logical world. Now suppose we have a model, M_1, of $\neg G_{\text{tcp}}$, and consider the submodel, M_2, containing just those objects m such that m satisfies at least one predicate in R that occurs in G_{tcp}. Note that M_2 is closed under (the interpretation of) every operation in S, because $\neg G_{\text{tcp}}$ implies that all of the "type-hypotheses" of G_{tcp} are true in M_1. This essentially excludes "bad atoms", as defined by the function `acl2::bad-atom`. Then because G_{tcp} is quantifier-free, M_2 also satisfies $\neg G_{\text{tcp}}$. We can turn M_2 into a model M_2' for the language of Z3, by assigning the appropriate type to every object. (As noted in Section 3.1, M_2' satisfies the theory of Z3 if M_2 is a model of ACL2(r); but for ACL2 that is not the case, so in future work, we expect to construct an extension of M_2' that satisfies all of the axioms for real closed fields.) Then we have the claim: for every assignment s from the free variables of G_{tcp} to M_2 with corresponding typed assignment s' from the free variables of G_{smt} to M_2' , if $\neg G_{\text{tcp}}$ is true in M_2 under s, then $\neg G_{\text{smt}}$ is true in M_2' under s'. Thus, if G_{tcp} is translatable, and $\neg G_{\text{smt}}$ is unsatisfiable, we conclude that G_{tcp} is a theorem in ACL2.

In the rest of this section, we discuss for each of the recognizer functions and each of the basic functions in ACL2, how we associate them with the corresponding Z3 functions.

3.1 Booleans, Integers, Rationals, and Reals

If a term is a boolean constant, then the translation to the SMT solver is direct. Likewise, if x_i is free in G_{tcp} and `(booleanp` x_i`)` is one of the hypotheses of G_{tcp}, then G_{tcp} holds trivially in the case that $x_i \notin \{\texttt{t}, \texttt{nil}\}$. Thus, in G_{smt} `Smtlink` can represent the hypothesis `(booleanp` x_i`)` with the declaration

```
x_i = Bool('x_i')
```

without excluding any satisfying assignments. We assume that the boolean operations of the SMT solver (e.g. `And`, `Or`, `Not`) correspond exactly to their ACL2 equivalents when their arguments are boolean. If a boolean operator is applied to a non-boolean value, then Z3 throws an exception, and we regard G as non-translatable.

Similar arguments apply in the case that x_i is an integer, rational, or real number. We represent ACL2 rational numbers as Z3 real numbers. Because every rational number is a real number, any satisfying assignment of rational numbers to rational variables in $\neg G_{\text{tcp}}$ has a corresponding assignment for $\neg G_{\text{smt}}$. Thus, G_{smt} is a generalization of G_{tcp}. We note that for ACL2, formally proving the soundness of this generalization requires extending our previously discussed M_2' model into a model that satisfies the theory of real closed fields [1], because we are translating rationals in ACL2 to reals in Z3. We haven't wrapped our heads around how to do that extension in a many-sorted setting, therefore we designate this to be future work. As with booleans, we assume that arithmetic and comparison operators have equivalent semantics in ACL2 and the SMT solver. In fact, we use the Python interface code to enforce this assumption. As an example, ACL2 allows the boolean values `t` and `nil` to be used in arithmetic expressions – both are treated as 0. Z3 also allows `True` and `False` to be used in integer arithmetic, with `True` treated as 1 and `False` treated as 0. To ensure that $G_{\text{smt}} \Rightarrow G_{\text{tcp}}$, our Python code checks the sorts of the arguments to arithmetic operators to ensure that they are integers or reals, where the interpretations are the same for both ACL2 and Z3.

When `Smtlink` is used with ACL2(r), non-classical functions are non-translatable. We believe that if $\neg G_{\text{tcp}}$ is classical and satisfiable, then there exists a satisfying assignment to $\neg G_{\text{tcp}}$ where all real-valued variables are bound to standard values. We believe the sketched proof in the beginning of Section 3 works well for ACL2(r). If we are wrong, we hope that one of the experts in non-standard analysis at the workshop will correct us.

3.2 Symbols

A very important basic type in ACL2 is `symbolp`. We represent symbols using an algebraic datatype in Z3. In the z3 interface class, we define a `Datatype` called `z3sym`, with a single field of type `IntSort`. Symbol variables are defined using the datatype `z3sym`. We then define a class called `Symbol`. This class provides a variable `count` and a variable `dict`. It also provides a function called `intern` for generating a symbol constant. This class keeps a dictionary mapping from symbol names to the generated `z3sym` symbol constants. This creates an injective mapping from symbols to natural numbers. All symbol constants that appeared in the term are mapped onto the first several, distinct, naturals.

If a satisfying assignment to $\neg G_{\text{tcp}}$ binds a symbol-valued variable to a symbol-constant that doesn't appear in G_{tcp}, then in our soundness argument, we construct a new symbol value for $\neg G_{\text{smt}}$ using an integer value distinct from the ones used so far – we won't run out. Thus, all symbol values in a satisfying assignment to $\neg G_{\text{tcp}}$ can be translated to corresponding values for $\neg G_{\text{smt}}$. The only operations that we support for symbols are comparisons for equality or not-equals. We assume that these operations have corresponding semantics in ACL2 and the SMT solver.

[1] http://smtlib.cs.uiowa.edu/theories-Reals.shtml

3.3 FTY types

We have added support for common `fty` types that enable `Smtlink` to automatically construct bridges from the untyped logic of ACL2 to the typed logic of Z3. Currently, `Smtlink` infers constructor/destructor relations and other properties of these types from the `fty::flextypes-table`. Thus, the use of `fty` types extends the trusted code to include the correctness of these tables. This trust is mitigated by two considerations. First, `Smtlink` only uses `fty` types that have been specified by the user in a hint to the `Smtlink` clause processor. If the user provides no such hints, no `fty` types are used by `Smtlink`, and no soundness concerns arise.

Second, we expect that the information that `Smtlink` obtains from these tables could be obtained instead from the ACL2 logical world using `meta-extract` in `Smtlink`'s verified clause-processor chain. We see the current implementation as a useful prototype to explore how to seamlessly infer type information from code written according to a well-defined type discipline.

3.3.1 fty::defprod

The algebraic datatypes of Z3 correspond directly to `fty::defprod`. `Smtlink` simply declares a Z3 datatype with a single constructor whose destructor operators are the field accessors of the product type. `Smtlink` requires that the arguments to the `fty` constructors satisfy the constructors' guards – otherwise G_{tcp} is non-translatable. The only operations on product types are field accessors, i.e. destructors. For translatable terms, the SMT type has the same construct/destructor theorems as the FTY type. Thus, the `Smtlink` translation maintains equivalence constructors and field accessors of product types.

3.3.2 fty::deflist

Lists are essentially a special case of a product type. For example,

Listing 1: ACL2 deflist

```
(deflist integer-list
  :elt-type integerp
  :true-listp t)
```

Listing 2: Z3 Datatype

```
integer_list= z3.Datatype('integer_list')
integer_list.declare('cons',
                     ('car', _SMT_.IntSort()),
                     ('cdr', integer_list))
integer_list.declare('nil')
integer_list = integer_list.create()
def integer_list_consp(l):
  return Not(l == integer\_list.nil)
```

ACL2 overloads `cons`, `car`, and `cdr` to apply to any list. In contrast, Z3's typed logic has a separate `cons`, `car`, `cdr` functions for each list type. This is why our examples from Section 4 require fixing functions to convey the type information to the trusted-clause processor. We believe that most of the users' burden of typed lists will be removed in a future release by adding type-inference to `Smtlink`. There are soundness issues that must be addressed. In ACL2, `(equal (car nil) nil)`. In Z3,

    integer_list.car(integer_list.nil)

is an arbitrary integer. To ensure soundness, the trusted-clause processor produces the proof obligation `(consp x)` for every occurrence of `(car x)` that it encounters. Under this precondition, the Z3 translation preserves the constructor/destructor relationship for lists. Likewise

    integer_list.cdr(integer_list.nil)

is an arbitrary `integer_list`. Thus, `Smtlink` enforces the `:true-listp t` declaration for list types.

Because “arbitrary” includes `integer_list.nil` in addition to all other `integer_lists`, these construction ensures that the SMT solver can choose the value for `integer_list.cdr(x̃)` for G_{smt} that corresponds to the value of `(cdr x)` for any assignments for G_{tcp}. The G_{smt} is a generalization of G_{tcp}.

3.3.3 fty::defalist

`Smtlink` represents alists with SMT arrays. We only support the operations `acons` and `assoc-equal` for alists. Then we have:

```
(defthm alist-axioms
  (implies (not (equal key1 key2))
           (and (equal (assoc key1 (acons key1 value alist)) value))
                (equal (assoc key1 (acons key2 value alist)) (assoc key1 alist)))
                (equal (assoc key1 nil) nil))
```

The corresponding theorem in the theory of arrays is

```
(defthm array-axioms
  (implies (not (equal addr1 addr2))
           (and (equal (load addr1 (store addr1 value array)) value))
                (equal (load addr1 (store addr2 value array)) (load addr1 array))))
```

Note that `Smtlink` does not support operations such as `cdr`, `nth`, `member`, or `delete-assoc` that would “remove” elements from an alist. Also, Z3 arrays are typed.

The key issue in the translation is how to handle the case when a key is not found in the alist (ACL2) or array (SMT). Our solution is to make the element type of the SMT array be an option type called `keyType_elementType`. This type can either be a `(key, value)` tuple or `keyType_-elementType.nil`. Thus, any value returned by `assoc-equal` with proper alist and key types has a corresponding `keyType_elementType` value. Thus, any value for an `assoc-equal` terms in G_{tcp} can be represented in G_{smt}.

When applying `cdr` to a `keyType_elementType`, we must ensure that the `keyType_elementType` value is not nil. This is analogous to the issue with lists: in ACL2, `(equal (cdr nil) nil)` but in Z3,

`keyType_elementType.cdr(keyType_elementType.nil)`

is an arbitrary value of `elementType`. Thus, the trusted-clause processor produces the proof obligation for ACL2 `(not (null x))` for every occurrence of `(cdr x)` when `x` is the return value from `assoc-equal`. Under this precondition, `cdr` is only applied to non-nil values from `assoc-equal` and we maintain correspondence of values for terms in G_{tcp} and G_{smt}.

By only providing `acons` and `assoc-equal`, the `Smtlink` support for alists is rather limited. Nevertheless, we have found it to be very useful when reasoning about problems where alists are used as simple dictionaries.

3.3.4 fty::defoption

As is shown in Program 3.3.4, the translation of `defoption` is straightforward.

Listing 3: ACL2 deflist

```
(defoption maybe-integer
           integerp)
```

Listing 4: Z3 Datatype

```
maybe_integer= z3.Datatype('maybe_integer')
maybe_integer.declare('some', ('val', IntSort()))
maybe_integer.declare('nil')
maybe_integer = maybe_integer.create()
```

In this example, the `maybe-integer-p` recognizer maps to `maybe_integer` type. The constructor `maybe-integer-some` maps to `maybe_integer.some`. The destructor `maybe-integer-some->val` maps to `maybe_integer.val`. The none type `nil` maps to `maybe_integer.nil`. Typical users of FTY types won't write `maybe-integer-some` constructor and `maybe-integer-some->val` destructors. They will first check if a term is nil, and then assume the term is an `integerp`. When a program returns an `integerp`, ACL2 knows it is also a `maybe-integerp`. Due to lack of type inference capabilities, `Smtlink` currently requires the user to use those constructors, therefore maintaining the option type through function calls where a `maybe-integerp` is needed and use the destructors where an `integerp` is needed. The constructor function satisfies the same theorems in ACL2 and Z3. Therefore, it's sound. For the field-accessor, when trying to access field of a `nil`, ACL2 returns the fixed default value, while Z3 will return arbitrary value of that `some` type. The Z3 values include the ACL2 value. `nil` is trivially the same. Thus, the `Smtlink` translation maintains equivalence of values of terms for constructors and field accessors of option types.

3.4 Uninterpreted Functions

The user can direct `Smtlink` to represent some functions as uninterpreted functions in the SMT theories. Let `(f arg1 arg2 ...argk)` be a function instance in G_{tcp}. `Smtlink` translates this to

```
    f_smt(arg_smt1, arg_smt2, ..., arg_smtk)
```

The constraints for `f_smt` are the types of the argument, the type of the result, and any user-specified constraints. If the function instance in G_{tcp} violates the argument type constraints, then the term is untranslatable – currently, `Smtlink` produces an SMT term that provokes a `z3types.Z3exception`. For each function instance in G_{tcp} that is translated to an uninterpreted function, `Smtlink` produces a proof obligation for ACL2 that the function instances in G_{tcp} satisfy the given type-recognizers. Likewise, if the user specified any other constraints for this function, they are returned as further ACL2 proof obligations. Under these preconditions, any value that can be produced by `f` satisfies the constraints for `f_smt`. Thus, we maintain correspondence of values for terms in G_{tcp} and G_{smt}.

4 A Ring Oscillator Example

In order to show the power of `Smtlink`, especially what benefit FTY types bring us, in this section, we take a look at how `Smtlink` can be used in proving an invariant of a small circuit. This example uses extensively the FTY types, including product types, list types, alist types, and option types. The simple circuit we want to model is a 3-stage ring oscillator. A ring oscillator is an oscillator circuit consisting of an odd number of inverters in a ring as is shown in Figure 3. A 3-stage ring oscillator consists of three inverters. It oscillates to provide a signal of a certain frequency.

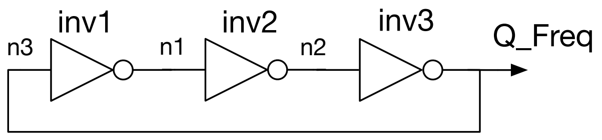


Figure 3: 3-Stage Ring Oscillator

For each inverter in this circuit, we say it is stable if its input is not equal to its output, otherwise, we say the inverter is ready-to-fire. One interesting invariant of this circuit is defined in Theorem 4.1:

Theorem 4.1 *Starting from a state where there is one and only one inverter ready-to-fire, for all future states, the ring oscillator will stay in a state where there is only one inverter ready-to-fire.*

In order to prove this property of this ring oscillator, we will discuss how we used FTY types extensively for circuit modeling and how `Smtlink` helped greatly at proving the theorem. To check the details of the proof, see projects/smtlink/examples/ringosc.lisp

4.1 Circuit and Trace Modeling using FTY Types

We model an **inverter** gate by defining a product type with two fields – `input` and `output`. We then model the 3-stage **ring oscillator** using a product type with six fields – three internal nodes `n1` through `n3` and three submodule inverters `inv1` through `inv3`. We call it `ringosc3-p`. We then define a **connection function** specifying the connections between the upper-level ring oscillator nodes and the lower-level inverter nodes. This describes the shape of the circuit.

As for modeling the behavior of the circuit, we use traces [7]. We define a circuit **state** as an alist mapping from signal names to its values. A **trace** is a list of circuits states, called `any-trace-p`. We define a **step recognizer** for an inverter. This recognizer function takes two consecutive steps from a trace as inputs and serves as a constraint function of what are the allowed behaviors in a step for an inverter. A valid trace for an inverter is defined recursively using the step function. A valid trace for the 3-stage ring oscillator is then defined requiring the trace to be valid for all three inverters. We call the recognizer function for a valid trace of a ring oscillator, `ringosc3-valid`.

We define a **counting** function for an inverter. The counting function returns 1 when an inverter is ready-to-fire and 0 when it's stable. Then we define a counting function for the ring oscillator based on the counting function for an inverter. We say a state of the ring oscillator is *one-safe* if only one inverter is ready-to-fire. We call the function `ringosc3-one-safe-state`. Using this function, we can define `ringosc3-one-safe-trace`, which means all states in a trace are *one-safe*. Theorem 4.1 is defined as Program 4.1.

Program 4.1 The ringosc3-one-safe theorem

```
(defthm ringosc3-one-safe
  (implies (and (ringosc3-p r)
                (any-trace-p tr)
                (consp tr)
                (ringosc3-valid r tr)
                (ringosc3-one-safe-state r (car tr)))
           (ringosc3-one-safe-trace r tr))
  :hints (("Goal"
           :induct (ringosc3-one-safe-trace r tr)
           :in-theory (e/d (ringosc3-one-safe-trace
                            ringosc3-valid
                            inverter-valid)
                           (ringosc3-one-safe-lemma)))
          ("Subgoal *1/1.1"
           :use ((:instance ringosc3-one-safe-lemma
                            (r r)
                            (tr tr)))
           )))
```

In this theorem, `Smtlink` helped to prove the inductive step. Due to space constraints, the details of `ringosc3-one-safe-lemma` are elided in this paper.. We note that proving this theorem using just ACL2 requires proving detailed lemmas about possible transitions of the ring oscillator. More specifically, our trace model requires asking if a signal exists in the state table, which causes a huge amount of case splits. However, this has not been a problem for `Smtlink` and the large amount of cases is handled efficiently. Using `Smtlink`, we are able to expand the functions out to proper steps and then prove the theorem without much human intervention. This demonstrates the potential of applying `Smtlink` to systems and proofs about systems. All this is made convenient because of the new theories supported and the useful user-interface.

5 Related Work

Integrating external procedures like SAT and SMT solvers into ACL2 has been done in several works in the past. Srinivasan [25] integrated the Yices [8] SMT solver into ACL2 for verifying bit-level pipelined machines. They use a trusted clause processor with a translation process. They appear to have mostly used the bit-vector arithmetic and SAT solving capabilities of Yices. Prior to that, in [17], they integrated a decision procedure called UCLID [16] into ACL2 to solve a similar problem. These are works that require fully trusting the integration.

A typical way of ensuring soundness and avoiding trusting external procedures is to followed Harrison and Théry's "skeptical" approach [12] and reconstruct proofs in the theorem prover. Recent work by Luís [13] allows refutation proofs of SAT problems to be reconstructed and replayed inside of ACL2. Their work focused on generating efficient refutation proofs that can be checked by a theorem prover in a short amount of time. Integrating SMT solvers into theorem provers has been a consistently developing area in the past decade [18, 11, 4, 19, 6, 5, 1, 10]. Erkök [10] integrated the SMT solver Yices into Isabelle/HOL. Similar to `Smtlink`, they not only have basic theories but also support algebraic datatypes. They trust Yices as an oracle. Works like [5, 9] do proof reconstruction. Sledgehammer [5, 14] is a proof assistant that integrates a bunch of SMT solvers into the theorem prover Isabelle/HOL. Proof reconstruction task is hard, and as pointed out in [14] the reconstruction can fail, and sometimes take a tremendous amount of time. Armand [1, 9] developed a framework in the theorem prover Coq for integrating external SMT solver results. They developed a set of "small checkers" that are able to take a certificate and call corresponding small checker for proof checks and used Coq tactics for automation. They report having achieved better performance than [5]. Also working on bridging the gap between interactive theorem proving and automated theorem proving (aka solvers), Moura [21] chooses a different path to build a theorem prover called Lean which also uses Z3, the SMT solver, as a back-end, but also provides benefits of an interactive theorem prover.

Several papers showed how their methods could be used for the verification of concurrent algorithms such as clock synchronization [11], and the Bakery and Memoir algorithms [19]. Erkök [10] uses the integration to prove memory safety of small C programs. [11] used the CVC-Lite [2] SMT solver to verify properties of simple quadratic inequalities. SMT solvers have drawn interests in the programming language research where [26] integrates SMT solvers into Haskell for proving refinement types.

Our work is based on our previous work [22]. Previously we showed how the integration of SMT solvers with theorem provers can help to prove properties of Analog/Mixed-Signal Designs. We used a single trusted clause-processor like is done in [25]. This paper describes our recent work of re-architecting `Smtlink` to verify the majority of it using verified clause-processors, therefore greatly improved soundness. We now depend on a very minimal core in a trusted clause-processor. In addition to that, our added support for algebraic datatypes largely broadened the horizon of problems `Smtlink` is able to handle.

6 Conclusion and Future Work

In this paper, we discussed an updated `Smtlink`. Comparing to the previous version, the current `Smtlink` has a more compelling argument of soundness, is extensible, and supports more theories. The architecture of `Smtlink` now clearly separates into verified and trusted parts. We make the trusted core as small as possible. We outlined an approach for verifying soundness, explaining how the gap is bridged between logic of ACL2 and the logic of an SMT solver. The new architecture through a sequence of verified clause-processors makes it extensible and easy to maintain. There are still many aspects we want to keep working on.

First, we want to use the meta-extract [15] capability introduced in last year's workshop. We believe if we can use the meta-extract to fully verify several of our clause-processors, for example, the function expansion clause processor, then the clause-processor won't need to return the clause back to ACL2 for proof. We observe that when projects get larger, the auxiliary clauses can become hard to prove, and making the clause-processor fully verified will reduce time spent on proving the auxiliary clauses to 0.

Second, given that we have the meta-extract capability, we are wondering if we can make a verified type inference engine. Currently, `Smtlink` knows nothing about types of terms and replies on the user for type instrumentation. Fixing functions have to be added to places where such a type information is required. For example, when dealing with a `nil`, which type of `nil` is it? We would love to deduce the types of terms and relieve the burden on the users for specifying types.

Third, we sketched a soundness proof in this paper, but this proof is not complete. Specifically we need to extend the model M_2' as described in Section 3 to a model that satisfies the real-closure axioms in a many-sorted setting. The current soundness proof sketch works well with ACL2(r) but not ACL2. We'd like to complete this proof.

Fourth, we want to make all counter-examples evaluable. This will require knowing FTY types and how to translate their constructors back. We also have to figure out how to translate algebraic numbers.

Fifth, we note `Satlink` has implemented a proof reconstruction interface that allows proofs to be returned from an SAT solver and replayed in ACL2. This can make the single trusted clause-processor goes away and remove all trusts we give to external SMT solvers. Proof reconstruction is an interesting direction that we might want to research more.

For applications, we believe the current `Smtlink` have enough capability that it can be applied to a lot of problems. We have been working on using it to verify properties of an asynchronous FIFO. The results are promising. In the future, we want to use it to prove timing properties making use of its arithmetic reasoning ability.

Acknowledgments

We would like to thank the ACL2 community for all the help in answering our questions while using ACL2 and developing `Smtlink`. We especially want to thank Matt Kaufmann for teaching us model theory and the many discussions to form a sketch of the soundness proof for `Smtlink`. We are also thankful to the anonymous reviewers for the insightful and constructive feedback.

References

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

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

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

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

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

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

[7] David L. Dill (1987): *Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits*. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, USA. Available at `http://reports-archive.adm.cs.cmu.edu/anon/scan/CMU-CS-88-119.pdf`. AAI8814716.

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

[9] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds & Clark Barrett (2017): *SMTCoq: A Plug-In for Integrating SMT Solvers into Coq*. In Rupak Majumdar & Viktor Kunčak, editors: *Computer Aided Verification*, Springer International Publishing, Cham, pp. 126–133, doi:10.1007/978-3-319-63390-9_7.

[10] Levent Erkök & John Matthews (2008): *Using Yices as an Automated Solver in Isabelle/HOL*. In: *In Automated Formal Methods08*, ACM Press, pp. 3–13. Available at `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.156.8123`.

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

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

[13] Marijn Heule, Warren Hunt, Matt Kaufmann & Nathan Wetzler (2017): *Efficient, Verified Checking of Propositional Proofs*. In Mauricio Ayala-Rincón & César A. Muñoz, editors: *Interactive Theorem Proving*, Springer International Publishing, Cham, pp. 269–284, doi:10.1007/978-3-319-66107-0_18.

[14] L. Paulson J. Blanchette (2017): *Sledgehammer*. `https://isabelle.in.tum.de/dist/doc/sledgehammer.pdf`. [Online; accessed 14-July-2018].

[15] Matt Kaufmann & Sol Swords (2017): *Meta-extract: Using Existing Facts in Meta-reasoning*. In Slobodová & Jr. [24], pp. 47–60, doi:10.4204/EPTCS.249.4. Available at `http://arxiv.org/abs/1705.00766`.

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

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

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

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

[20] Leonardo Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In C.R. Ramakrishnan & Jakob Rehof, editors: *Tools and Algorithms for the Construction and Analysis of Systems*, *Lecture Notes in Computer Science* 4963, Springer Berlin Heidelberg, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[21] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn & Jakob von Raumer (2015): *The Lean Theorem Prover (System Description)*. In Amy P. Felty & Aart Middeldorp, editors: *Automated Deduction - CADE-25*, Springer International Publishing, pp. 378–388, doi:10.1007/978-3-319-21401-6_26.

[22] Yan Peng & Mark Greenstreet (2015): *Extending ACL2 with SMT Solvers*. In Matt Kaufmann & David L. Rager, editors: *Proceedings Thirteenth International Workshop on the ACL2 Theorem Prover and Its Applications*, *Electronic Proceedings in Theoretical Computer Science* 192, Open Publishing Association, Austin, Texas, USA, 1–2 October 2015, pp. 61–77, doi:10.4204/EPTCS.192.6.

[23] Yan Peng & Mark Greenstreet (2015): *Integrating SMT with Theorem Proving for Analog/Mixed-Signal Circuit Verification*. In Klaus Havelund, Gerard Holzmann & Rajeev Joshi, editors: *NASA Formal Methods*, Springer International Publishing, pp. 310–326, doi:10.1007/978-3-319-17524-9_22.

[24] Anna Slobodová & Warren A. Hunt Jr., editors (2017): *Proceedings 14th International Workshop on the ACL2 Theorem Prover and its Applications, Austin, Texas, USA, May 22-23, 2017*. *EPTCS* 249. Available at `http://arxiv.org/abs/1705.00766`.

[25] S.K. Srinivasan (2007): *Efficient Verification of Bit-level Pipelined Machines Using Refinement*. Ph.D. thesis, Georgia Institute of Technology. Available at `http://hdl.handle.net/1853/19815`.

[26] Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler & Ranjit Jhala (2017): *Refinement Reflection: Complete Verification with SMT*. *Proc. ACM Program. Lang.* 2(POPL), pp. 53:1–53:31, doi:10.1145/3158141.

DefunT: A Tool for Automating Termination Proofs by Using the Community Books (Extended Abstract)

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

We present a tool that automates termination proofs for recursive definitions by mining existing termination theorems.

The macro, `defunT` (**defun** with auto-**T**ermination), is a tool that can automate ACL2 proofs of termination (i.e., of measure conjectures). This note has three goals: to introduce this tool to potential users, to explain some of its implementation, and to advertise for research collaborators to improve the tool. The tool suite resides in community books directory `kestrel/auto-termination/`.[1]

`DefunT` relies on a database of already-proved termination theorems, each stored as a list of clauses (disjunctions). That database is generated by the script file `write-td-cands.sh`, which writes it to the generated book, `td-cands.lisp`, and creates an associated file, `td-cands.acl2`. This script computes the database after it includes the book `doc/top.lisp`, which in turn includes many of the community books (to build the manual), using algorithms implemented in the book, `termination-database.lisp`. The book `td-cands.lisp` will likely only need to be regenerated infrequently; but it is routinely certified by the build system on top of a world obtained by executing the 45 `include-book` events in `td-cands.acl2`, which define all necessary packages so that ACL2 can read all forms in the book.

We explain `defunT` — both its use and a little about its implementation — by focusing on the following example, which creates three distinct proof goals for termination: one for each recursive call. The book `defunt-top.lisp` includes both the database, `td-cands.lisp`, and the implementation of the `defunT` macro, `defunt.lisp`.

```
(include-book "kestrel/auto-termination/defunt-top" :dir :system)
(defunt f3 (x y)
  (if (consp x)
      (if (atom y)
          (list (f3 (cddr x) y) (f3 (cadr x) y))
        (f3 (cdr x) y))
    (list x y)))
```

The output shown below notes that `defunT` finds three helpful termination theorems in the database, `td-cands.lisp`. Each of these suffices to prove one of the three goals with a `:termination-theorem` lemma-instance, where one of those three requires a book to be included.

```
*Defunt note*: Using termination theorems for SYMBOL-BTREE-TO-ALIST-AUX,
EVENS and TRUE-LISTP.

*Defunt note*: Evaluating
(LOCAL (INCLUDE-BOOK "misc/symbol-btree" :DIR :SYSTEM))
to define function SYMBOL-BTREE-TO-ALIST-AUX.
```

[1] An archival version, from the time this paper was written, is under `books/workshops/2018/kaufmann/`.

The `defunT` macro uses `make-event` to do the search and to generate a suitable event as displayed below. The search can make two passes through the database, where the first pass only considers functions defined in the current session. In this example, a local `include-book` is generated because the first pass was not sufficient. In spite of making both passes, ACL2 reports only 0.04 seconds taken altogether, using a 2014 MacBook Pro.

```
(ENCAPSULATE ()
  ;; The following book is necessary, as noted in the output shown above.
  (LOCAL (INCLUDE-BOOK "misc/symbol-btree" :DIR :SYSTEM))
  ;; Six local defthm events are omitted here.  The seventh has the following form:
  (LOCAL (DEFTHM new-termination-theorem
           <termination theorem for f3 generated from found measure etc.>
           :HINTS (("Goal" :USE <..elided here..> :IN-THEORY (THEORY 'AUTO-TERMINATION-FNS)))))
  (DEFUN F3 (X Y)
    (DECLARE (XARGS :MEASURE (ACL2-COUNT X)
                    :HINTS (("Goal" :BY (:FUNCTIONAL-INSTANCE new-termination-theorem
                                                              (binary-stub-function F3))))))
    (IF (CONSP X)
        (IF (ATOM Y)
            (LIST (F3 (CDDR X) Y) (F3 (CADR X) Y))
            (F3 (CDR X) Y))
        (LIST X Y))))
```

A key aspect of `defunT` is that termination theorem clause-lists are stored in *simplified* form: thus, an old clause-list can subsume a new clause even when function bodies have minor differences, such as `(if (endp x) ...)` vs. `(if (not (consp x)) ...)`. Also, the generated local theorems are carefully instrumented to make proofs fast and automatic. The flow is as follows (here, restricting to the case of a single old termination theorem), where *old* and *new* are old and new termination theorems, and old_s and new_s are their simplifications: *new* follows with a `:use` hint from new_s, which follows with a `:by` hint from old_s, which follows with a `:use` hint from *old*. The `:by` hint has two advantages over a corresponding `:use` hint: it avoids the need to supply a substitution (when the old and new functions have different formals), and it avoids if-splitting into clauses (goals). The `:by` hint succeeds because it employs essentially the same subsumption test as is used during the search for an old termination theorem to prove the new termination goal. The `:use` hints are accompanied by `:in-theory` hints that can be expected to make those proofs fast, by restricting to the small theories used for clause-list simplification. Stub functions replace functions called in their own termination schemes, to enhance subsumption.

Conclusion. Program termination is a rich field [1]. The goal of `defunT` is, however, simply to make it convenient to prove termination automatically when using ACL2. An extension of ACL2 with CCG analysis [2] can prove termination automatically; unlike that approach, `defunT` generates a measure for ACL2's usual termination analysis. J Moore's tool Terminatricks [3] is a different step towards that goal: while that tool does not use the `defunT` approach of taking advantage of the community books, it can however incrementally extend its database of termination theorems. This potential enhancement to `defunT` is discussed in file `to-do.txt`, as are more than 20 others. Further implementation-level details may be found in the `README`, which for example explains database organization by *justification* (which includes a measure), as well as several optimizations, such as the use of subsumption to restrict the database to 643 distinct termination schemes essentially shared by 821 functions. Others are invited to contribute to the enhancement of `defunT`!

Acknowledgments. This material is based upon work supported in part by DARPA under Contract No. FA8750-15-C-0007. We thank Eric Smith for encouragement to work on this problem.

References

[1] Byron Cook, Andreas Podelski & Andrey Rybalchenko (2011): *Proving Program Termination*. *Commun. ACM* 54(5), pp. 88–98. Available at `http://doi.acm.org/10.1145/1941487.1941509`.

[2] Matt Kaufmann, Panagiotis Manolios, J Strother Moore & Daron Vroon (2006): *Integrating CCG analysis into ACL2*. In: *Workshop Proceedings: WST 2006, Eighth International Workshop on Termination*, pp. 64–68. Available at `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.97.8994&rep=rep1&type=pdf`.

[3] J Moore (Accessed: 2018): *Terminatricks*. `https://github.com/acl2/acl2/tree/master/books/projects/codewalker/terminatricks.lisp`.

Hint Orchestration Using ACL2's Simplifier

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

This paper describes a strategy for providing hints during an ACL2 proof, implemented in a utility called `use-termhint`. An extra literal is added to the goal clause and simplified along with the rest of the goal until it is stable under simplification, after which the simplified literal is examined and a hint extracted from it. This simple technique supports some commonly desirable yet elusive features. It supports providing different hints to different cases of a case split, as well as binding variables so as to avoid repeating multiply referenced subterms. Since terms used in these hints are simplified in the same way as the rest of the goal, this strategy is also more robust against changes in the rewriting normal form than hints in which terms from the goal are written out explicitly.

1 Introduction

ACL2's mechanism for giving hints to the prover is very powerful and general. Users can provide arbitrary code to evaluate whether a hint should be given and what that hint should be. However, it is surprisingly difficult to accomplish certain things with computed hints. We notice that a frequent stumbling block is the difficulty of predicting the exact form of some term. A user may know the initial form of the term but fail to predict its normal form under rewriting; it may even rewrite to different terms in different cases.

Consider a proof where we want to supply different hints to different cases. Suppose the proof splits into two cases, one assuming `A` and one assuming `(not A)`, where `A` is some term. Say we want to supply a hint `H1` for the case where `A` is assumed true and `H2` for the other case. Users often provide subgoal hints in such cases, even though these can be broken either by changes to the relevant functions or by changes to ACL2 system heuristics. Alternatively, a computed hint can examine the clause to see when the case split occurs and which case has resulted—if `(not A)` is a member of the clause, then `A` has been assumed true, and if `A` is a member of the clause, it is assumed false:

```
  :hints ((and (member-equal '(not A) clause) H1)
          (and (member-equal 'A clause) H2))
```

However, a small change in rewriting strategy could cause the exact form of the term `A` in the clause to change, in which case neither hint will fire.

A second kind of problem occurs very frequently when doing inductive proofs and giving a hint to expand a conclusion term. The hint may work in many cases, but frequently there is at least one case where some variable is substituted for a known value, causing the hint to fail. Consider the following example:

```
(defund add-to-lst (x n)
  (if (atom x)
      x
    (cons (+ n (car x))
         (add-to-lst (cdr x) n))))

(defthm len-of-add-to-lst
  (implies (true-listp x)
           (equal (len (add-to-lst x n))
                  (len x)))
  :hints (("goal" :induct t
           :expand ((add-to-lst x n)))))
```

This proof fails because in the base case where `x` is an atom, it is known to be `NIL` due to the `true-listp` assumption. This substitution occurs before `(add-to-lst x n)` is expanded, and then the expand hint doesn't match.

We describe a computed hint utility that addresses this problem by letting ACL2 simplify the terms that will be put into the hint as it is simplifying the goal itself. This mechanism allows the user to give different hints when considering different cases, and the manner of giving these hints is idiomatic—simply write a term that produces different hints under different `if` branches. It allows terms to be bound to variables and used multiple times, and since the terms that will appear in the hints are translated and simplified by ACL2 before the hint is produced, the user does not need to provide them as translated terms or in simplified normal form in order to match other occurrences of these terms in the goal.

In Section 3 we describe what the utility does and give an example of how it is used. In Section 4 we explain the rationale behind certain design decisions, and in Section 5 we describe an extension to the utility that supports sequencing hints in multiple phases of a proof. In Section 6 we describe a proof effort in which this mechanism was used a great deal and give an extended example explaining one theorem proved using the utility.

2 Related Work

All major interactive theorem provers allow proofs to be structured at a high level by proving lemmas that may then be used in later proofs. They differ more in their approaches to proving individual lemmas. ACL2 has a robust default proof strategy that the user can modify by giving hints [1]. HOL and Coq proofs can be constructed directly at a very low level or may be directed by giving commands called *tactics* and combining them using tactic combinators called *tacticals* [3][2]. Isabelle also has tactics, but users more commonly direct proofs by structuring them using the Isar language, which supports outlining a proof using a syntax that is reminiscent of mathematical prose, but also contains directions on how to prove each subgoal [5].

The ACL2 Community Books [1] contains many libraries that provide special-purpose computed hints for reasoning in certain domains. However, there are few contributions that aid users in constructing their own hints. An exception is the community book `misc/computed-hint.lisp` written by Jun Sawada [4], which provides a set of practical utilities intended to improve computed hints. These support, for example, pattern matching against terms occurring in the clause while allowing substitution into the hint to be generated. We view these utilities as complementary to the mechanism described here.

3 Basic Usage

Our hint utility, `use-termhint`, is implemented in the ACL2 community book `std/util/termhints`. The utility is a computed hint form that takes a *hint term* provided by the user. Here is the definition of `use-termhint`:

```
(defmacro use-termhint (hint-term)
  `'(:computed-hint-replacement
     ((and stable-under-simplificationp
           (use-termhint-find-hint clause)))
     :use ((:instance use-termhint-hyp-is-true
            (x ,hint-term)))))
```

Initially, this just gives a `:use` hint and sets up another hint to fire when the goal is stable under simplification. The `:use` hint instantiates the theorem `use-termhint-hyp-is-true`, whose body is `(use-termhint-hyp x)`, where `use-termhint-hyp` is an always-true function for which no rules are enabled. The result of the `:use` hint is that a hypothesis `(use-termhint-hyp user-hint-term)` is added to the goal clause. Since no more hints are given until the goal is stable under simplification, this literal, including the hint term, is simplified along with the rest of the clause.

The computed-hint-replacement form produced by the above hint causes a second computed hint, `(use-termhint-find-hint clause)`, to fire when the clause is stable under simplification. This computed hint looks in the clause for a hypothesis that is a call of `use-termhint-hyp` and extracts a hint from its argument; it removes the `use-termhint-hyp` hypothesis using a custom-built clause processor and then issues the extracted hint. We describe how the hint is extracted in more detail below. First, here is an example of such a computed hint:

```
:hints ((use-termhint
         (let* ((f (foo a b))
                (g (bar f c))
                (h (baz f d))
                (i (fa g h)))
           (if (consp g)
              `'(:use ((:instance my-lemma
                        (x ,(hq g)) (y ,(hq h)) (z ,(hq i)))))
             `'(:expand ((fa ,(hq g) ,(hq h))))))))
```

When this hint initially fires, it places the `let*` term into the goal as the argument of a new hypothesis `(use-termhint-hyp ...)`. ACL2 then beta-reduces and simplifies this term along with the rest of the clause. The presence of this additional literal will cause a case split due to its `if` test. When the goal is stable under simplification, this term will have simplified into something derived from the `` `'(:use ...) `` subterm in one case and the `` `'(:expand ...) `` subterm in the other case. For the latter, the resulting term looks like this, if we assume that none of the `baz`, `bar`, or `foo` subterms were successfully rewritten:

```
(CONS
 (QUOTE QUOTE)
 (CONS (CONS (QUOTE :EXPAND)
             ...
             (HQ (BAR (FOO A B) C))
             ...
             (HQ (BAZ (FOO A B) D))
             ...)
       (QUOTE NIL)))
```

To extract the intended hint from this term, we use `process-termhint`, a simple term interpreter that understands `quote`, `cons`, and `binary-append` (since this can be produced when using `,@` inside backticks). It also treats the function `hq` the same as `quote`, which we explain in Section 4 below. This reduces the above term to the following hint:

```
'(:EXPAND ((FA (BAR (FOO A B) C)
               (BAZ (FOO A B) D))))
```

As a special case, if the hint term reduces to `NIL`, then no hint is given.

4 Design Decisions

Readers may note some oddities in the example above:

- Why `,(hq g)`, etc., rather than `,g`?
- Why use backquote-quote in `` `'(:use ...) `` rather than just backquote, i.e., `` `(:use ...) ``?

4.1 HQ

The use of `hq` distinguishes subterms that should not be interpreted by `process-termhint` but simply passed through as if quoted. We can't use `quote` directly because of its special meaning as a syntax marker rather than a function: `,(quote g)` would produce the symbol `g` instead of substituting the `let` binding of `g`. Using `hq` works because it is not treated specially by anything other than `process-termhint`. (In the logic, it is simply a unary stub function.)

We could define `process-termhint` differently to avoid needing to use `hq` in most cases: it could treat any term with leading function symbols other than `cons` and `binary-append` as quotations, in which case we could simply use `,g` instead of `,(hq g)` – but this wouldn't work if `g` was bound to a term that was (or simplified to) a call of `cons` or `binary-append`. Since we don't wish to require users to be cognizant of this difference between `cons` and `binary-append` and other function symbols, we decided instead to require that `hq` be used to quote terms that `process-termhint` should not interpret.

4.2 Backquote-Quote

To allow the most general usage of this tool, the hints passed to ACL2 from `use-termhint` are actually computed hints rather than literal keyword-value lists. We can think of `process-termhint` as evaluating the hint term (though it only allows `cons` and `binary-append` as function symbols); however, its result is then passed to ACL2's computed hint interpreter, which evaluates it again. That is why the examples above show hint keyword/value lists preceded by backquote-quote. If the quote occurring immediately after the backquote in the `` `'(:expand ...) `` was omitted, the result from `process-termhint` would be `(:expand ...)` instead of `'(:expand ...)`; the former would cause an error when evaluated again by ACL2's computed hint mechanism, since it is not a valid term, whereas the evaluation of the latter yields the expected keyword/value list. We actually support this slight abuse as a special case, adding a quote to any hint that would otherwise begin with a keyword symbol. But in the more general case, this double evaluation scheme allows hint terms to produce computed hints, as in the following example:

```
:hints ((use-termhint
          (let* ((q (foo a b)))
            `(my-computed-hint-function
              ',(hq q) clause id stable-under-simplificationp))))
```

For the common case where the hint term directly produces a keyword/value list, we support both the `` `'(:key0 val0 ...) `` form, which is doubly evaluated, and the simpler `` `(:key0 val0 ...) `` form, whose result after evaluation by `process-termhint` is quoted so as to nullify the second evaluation.

5 Sequencing Hints

It is sometimes useful to provide several stages of hints. We support this in the `use-termhint` utility via a macro `(termhint-seq hint-term1 hint-term2)`. This can be used inside a term passed to `use-termhint`. When simplifying the initial hint term in which it occurs, `hint-term1` will get simplified while `hint-term2` is wrapped in a call of `hide`, which prevents it from being simplified. Once the initial simplification is complete, the hint resulting from `hint-term1` is applied, and additionally, `hint-term2` is provided as the term to a new invocation of `use-termhint` with the `hide` removed.

A simple example:

```
:hints ((use-termhint
          (let ((a (bar f c)))
            (termhint-seq
              ''(:in-theory (enable my-theory1))
              (if (foo a b)
                  ''(:in-theory (enable my-theory2))
                ''(:in-theory (enable my-theory3)))))))
```

The `if` test in the second argument to `termhint-seq` does not cause a case split until after the first hint (enabling `my-theory1`) takes effect, because it is inside a `hide`. The `let` binding of `a` does apply to the occurrence of `a` in this term.

There is an unfortunate interaction between ACL2's function definition normalization feature and `termhint-seq` which that can occur when `termhint-seq` is used in a function. (A user might create a function rather than putting the whole term in the hint due to an aesthetic preference to have a theorem appear in its book without too large a hint list attached.) Normalization causes `if` tests to be pulled out of function calls, even for `hide`: `(hide (if a b c))` becomes `(if a (hide b) (hide c))`. In the above example, if the hint was defined in a function rather than given explicitly, the case split would occur before the first hint was given instead of after. This can be avoided by giving the declaration `(xargs :normalize nil)` when defining a function that uses `termhint-seq`.

6 Application

We used this utility frequently in a proof of the correctness of Tarjan's strongly connected components algorithm, accessible in the ACL2 community book `centaur/misc/tarjan.lisp`. Much of this proof involves technical lemmas about the existence of paths through the graph. For example, one usage of `use-termhint` is in the following theorem:

```
(defthm reachable-through-unvisited-by-member-cond-2
  (implies (and (tarjan-preorder-member-cond x preorder new-preorder)
                (graph-reachable-through-unvisited-p z y preorder)
                (not (graph-reachable-through-unvisited-p x y preorder)))
           (graph-reachable-through-unvisited-p z y new-preorder))
  :hints ((use-termhint (reachable-through-unvisited-by-member-cond-2-hint
                         x y z preorder new-preorder))))
```

We'll informally define enough here to say what the theorem means and how it is proved. First, `(graph-reachable-through-unvisited-p x y preorder)` says that there exists a path from graph node `x` to graph node `y` that does not include any member of `preorder`. Then, `(tarjan-preorder-member-cond x preorder new-preorder)` describes the final visited node set of a depth-first-search (DFS) when the DFS is run on a node `x` with initial visited node set `preorder`:

```
(defun-sk tarjan-preorder-member-cond (x preorder new-preorder)
  (forall y
         (iff (member y new-preorder)
              (or (member y preorder)
                  (graph-reachable-through-unvisited-p x y preorder)))))
```

I.e., a node `y` will have been visited by the time the DFS returns if either it was already visited before the DFS started, or it can be reached from `x` without traversing any already-visited nodes.

The theorem above shows that the visited nodes after a DFS starting from `x` don't break any paths to nodes that weren't reachable from `x`. That is, given a node `y` that is not reachable from `x` but is reachable from some node `z`, then it is still reachable from `z` when omitting the nodes that are newly visited after a DFS starting from `x`.

To prove this, we use the path from `z` to `y`. If that path is still valid after the DFS (i.e. it doesn't intersect `new-preorder`) then we're done. Otherwise, we'll use that path to construct a path from `x` to `y`. Since we assume the path does intersect `new-preorder`, let `i` be a witness to that intersection, i.e. a node in that path that is also in `new-preorder`. The suffix of the path starting at `i` is a path from `i` to `y` that does not intersect `preorder`. Additionally, since `i` is not in `preorder`, `tarjan-preorder-member-cond` implies `i` is reachable from `x` without intersecting `preorder`. Composing the path from `x` to `i` with the path from `i` to `y` yields a path from `x` to `y` not intersecting `preorder`, so we have contradicted our assumption that `y` is not reachable from `x`.

Here is the function producing the hint term used to prove the theorem via the chain of reasoning described above:

```
(defun reachable-through-unvisited-by-member-cond-2-hint
  (x y z preorder new-preorder)
  ;; Hyp assumes z reaches y in preorder.  Get the path from z to y:
  (b* ((z-y (graph-reachable-through-unvisited-canonical-witness z y preorder))
       ;; If that doesn't intersect the new preorder, then z reaches y
       ;; via that same path in the new preorder.
       ((unless (intersectp z-y new-preorder))
        `'(:use ((:instance graph-reachable-through-unvisited-p-suff
                  (x z) (visited new-preorder)
                  (path ,(hq z-y))))
           :in-theory (disable graph-reachable-through-unvisited-p-suff)))
       ;; Otherwise, get a node that is in both the path and the new-preorder
       (i (intersectp-witness z-y new-preorder))
       ((when (member i preorder))
        ;; this can't be true because it's a member of z-y
        `'(:use ((:instance intersectp-member
                  (a ,(hq i))
                  (x ,(hq z-y))
                  (y ,(hq preorder))))
           :in-theory (disable intersectp-member)))
       ;; Since i is a member of new-preorder it's reachable from x
       ((unless (graph-reachable-through-unvisited-p x i preorder))
        `'(:use ((:instance tarjan-preorder-member-cond-necc
                  (y ,(hq i))))
           :in-theory (disable tarjan-preorder-member-cond-necc)))
       ;; Construct the path from x to y using the paths from x to i and i to y
       (x-i (graph-reachable-through-unvisited-canonical-witness x i preorder))
       (x-y (append x-i (cdr (member i z-y)))))
    `'(:use ((:instance graph-reachable-through-unvisited-p-suff
              (path ,(hq x-y)) (visited preorder)))
       :in-theory (disable graph-reachable-through-unvisited-p-suff))))
```

The termhint utility is well suited to this sort of proof because there are many steps that would be difficult to automate: a rule that would find a correct witnessing path from `x` to `y` in the example above would need to be either very specific or very smart. Instead, we guide the proof at a high level by defining the case split and providing specific hints that resolve each of the cases separately.

A more common approach to this kind of proof development is to perform each step as its own lemma. These lemmas likely aren't suitable as general rules but can be instantiated to make process in the current proof. The theorem discussed above can be proved using four lemmas corresponding to the cases in the hint function. This is a reasonable approach, and having each case stand alone as a lemma might make it more clear how to debug any problems (though see below for a strategy to aid debugging proofs that use `use-termhint`). However, stating the lemmas requires repeating the applicable hypotheses and witness terms for each lemma individually, and this clutter makes it harder for a human to understand the chain of reasoning.

The main problem in debugging proofs that use `use-termhint` is determining which case generated a failed checkpoint. To remedy this, the book defining `use-termhint` provides an always-true function `mark-clause` and a corresponding theorem `mark-clause-is-true`. Used as follows, it adds a hypothesis `(mark-clause 'my-special-case)`:

```
  :use ((:instance mark-clause-is-true (x 'my-special-case)))
```

Adding such an instantiation to the hints produced by suspect cases effectively labels each resulting subgoal with a name that clarifies where it came from.

7 Conclusion

The `use-termhint` utility in some ways goes against the prevailing philosophy on how to prove theorems using ACL2. That is, when possible, it is better to avoid using hints to micromanage the prover, and instead to create rewriting theories that solve problems more automatically and robustly. But sometimes it is necessary to do a proof using a complicated sequence of reasoning steps that don't seem to be, in any obvious way, applications of nice rules. In such cases, `use-termhint` provides a robust, convenient, idiomatic method of structuring a proof at a high level and providing the needed hints.

References

[1] ACL2 Community (Accessed: 2018): *ACL2+Books Documentation*. Available at `http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/index.html`.

[2] The Coq Development Team (2018): *The Coq Proof Assistant Reference Manual*, 8.8 edition. Available at `http://coq.inria.fr/distrib/current/refman/index.html`.

[3] Michael Norrish & Konrad Slind (2018): *The HOL System Description*, 3rd edition. Available at `http://hol-theorem-prover.org`.

[4] Jun Sawada (2000): *ACL2 Computed Hints: Extension and Practice*. Technical Report TR-00-29, The University of Texas at Austin, Department of Computer Sciences. ACL2 Workshop 2000 Proceedings, Part A.

[5] Makarius Wenzel (2017): *The Isabelle/Isar Reference Manual*. Available at `http://isabelle.in.tum.de/dist/Isabelle2017/doc/isar-ref.pdf`.