Making Induction Manifest in Modular ACL2 *

Carl Eastlund Matthias Felleisen

Northeastern University
Boston, Massachusetts, U.S.A.
{cce,matthias}@ccs.neu.edu

Abstract

ACL2, a Common Lisp-based language for programming and theorem proving, has enjoyed industrial success despite lacking modern language features such as a module system. In previous work, we equipped ACL2 with modules, interfaces, and explicit linking and measured our system with a series of experiments. One experiment revealed a serious lack of expressivity; the interfaces cannot describe the induction schemes necessary to reason about exported functions with nontrivial patterns of recursion.

In this paper we revise our language, Modular ACL2, to overcome this weakness. The first novelty is the inclusion of manifest function definitions in interfaces from which ACL2 can infer induction schemes. The second novelty consists of the first proofs of soundness and expressivity for Modular ACL2; we also reaffirm the usefulness of our system with updated benchmarks.

Categories and Subject Descriptors D [*2*]: 2—Modules and interfaces; D [*2*]: 4—Formal methods; D [*3*]: 3—Modules, packages; F [*3*]: 1—Mechanical verification

General Terms Languages, Verification.

1. Programs and Proofs

Over the past two decades, a number of industrial labs have adopted ACL2 as a primary tool. The ACL2 system combines a purely functional, first-order subset of Common Lisp with the latest incarnation of the Boyer-Moore theorem prover [12, 13]. Roughly speaking, it extends first-order logic with axioms based on the function definitions in a program. Industrial programmers use ACL2 primarily as a modeling language for hardware and low-level software components. A typical usage pattern is to model the component as an ACL2 program, to validate the model with the (very large) preexisting test suite for the component, and then to prove the desired theorems about the component via the model.

Four years ago, we started supporting Rex Page's educational project of training software engineering students in high-assurance methods with ACL2 [21]. Page's year-long course sequence introduces students to unit testing, integration testing, random testing, and theorem proving—all available in Dracula, our dialect of ACL2 [22, 26]. While Page's first course focuses on small, single-programmer projects, the second course applies these techniques in a team-programming context.

The problem for both industrial and educational uses of ACL2 is that its programming language lacks a modern module system, and the theorem prover doesn't support modular reasoning. Modules and interfaces provide scope, abstraction, and specification boundaries, as well as reusable components. These principles are especially important for ACL2. Without abstraction boundaries, the theorem prover's search space grows as each new rule is admitted. Without specifications, it is hard to predict the effect of adding a new component; the lack of lexical scope causes unnecessary name clashes between components.

Currently, ACL2 relies on the package constructs it inherits from Common Lisp [24] and methods for proof encapsulation and functional instantiation, which provide abstraction and reusability. Packages are cumbersome and not lexically scoped; encapsulation comes at the cost of executability. As a result, researchers as well as students have to learn usage patterns to manually mimic modular organizations of programs and proofs.

Since manually maintaining these patterns is a laborious and error-prone process, we have added a module system to Dracula [5]. The module system takes its inspiration from ML's functor system [10, 16]; it separates modules from interfaces and introduces an external linking language. Naturally we impose enough restrictions to ensure the soundness of the ACL2 theorem prover, which assumes a first-order, terminating programming language. Our initial report also includes a number of benchmarks, i.e., attempts to turn modular systems into "monolithic" programs and to measure the effect on theorem-proving time.

The benchmarks show remarkable performance improvements introduced by our module system, in some cases by several orders of magnitude. Sadly, they also prove that it can only successfully reason about list-processing functions. For other algorithms, ACL2 fails to extract the proper induction scheme, rendering the module system useless. Furthermore, our original module system comes without a soundness proof; users must simply trust our reasoning.

In this paper, we enhance the module system with *manifest functions*, allowing the specification of induction schemes in interfaces exactly as in ACL2 programs. We also supply a formal soundness proof, establishing the correctness of our verification process. This is followed by a proof of expressivity, showing that with the addition of manifest functions we can modularize existing ACL2 proofs by splitting them at arbitrary boundaries. Finally, we complete our benchmarks for the previously problematic tasks, confirming the benefits of Modular ACL2.

The rest of the paper starts with a discussion of related work in section 2. We follow this with a brief introduction to ACL2 in section 3. Next we present Modular ACL2 and illustrate the problem with our first formulation in section 4. Section 5 presents our updated module system, with a formal semantics for verification

* This research was partially supported by several grants from the National Science Foundation.

PPDP'09, September 7–9, 2009, Coimbra, Portugal.
Copyright © 2009 ACM 978-1-60558-568-0/09/09...$10.00

and execution, followed by a proof of soundness in section 6. We describe our implementation and show updated benchmarks in section 7 and finally present our conclusions in section 8.

2. Related Work

The design space of modules with explicit specifications and external linking has been well-explored. The literature begins with Modula-2 [27], and includes more recent developments such as the ML module system [10, 16], PLT Scheme units [9, 19], and mixin modules [4]. These systems introduce a variety of features, including higher-order and first-class modules, recursive linking, and opaque, translucent, and transparent specifications.

It is well-known that theorem provers impose somewhat different requirements on module systems than regular programming languages. The simplest modular constructs for theorem provers include Isabelle's locales [11], Coq's sections [25], and the "little theories" of IMPS [6]. These are little more than lightweight scope and abstraction mechanisms; they can be used to separate parts of a proof, and their abstract local definitions can be instantiated to extract variations on their exports. So long as the underlying logic can express higher order abstractions, these systems do not extend its expressivity, but instead provide syntactic convenience.

Extended ML (EML) [23] equips SML [18] with logical properties and a verification semantics. The language is designed around a methodology of beginning with an abstract specification and refining it step-by-step to a concrete implementation. EML allows the user to supply the term "**?**" for any type, value, or structure (module), representing a component whose implementation is deferred but assumed correct. This allows a top-down development style in which there may be no executable implementation until the very end, but individual proof fragments can be checked along the way.

Coq [2, 1] inherits aspects of the ML module system and enriches it with a language of logical specifications. These are expressive enough to describe the specific implementation of a term (value or type), much like the manifest type specifications of ML. In turn, manifest type specifications allow the client of a specification to reason about the precise definition of an imported term.

Our module system for ACL2 inherits many aspects of these prior systems and builds on them. We use modules with external, hierarchical linking specifications much like units in PLT Scheme, implicit merging of abstract and concrete definitions when linking from mixin modules, and external, translucent interfaces combining opaque and transparent (manifest) specifications in the manner of ML. Modular ACL2 also supports a top-down development process for ACL2, similar to Extended ML. The introduction of parameterized components allows low-level details to be left as an abstract import while high-level parts of the proof are developed.

Furthermore, our system provides higher-order, instantiable abstractions much like locales, sections, "little theories", and the functors of ML-like systems. Unlike other theorem provers, ACL2 does not express higher-order abstractions natively. We have to synthesize a method for expressing, verifying, and instantiating abstract proofs within a first-order logic.

Our new version of Modular ACL2 shares with Coq the power to provide concrete specifications to express induction schemes at module boundaries. These specifications play a greater pragmatic role in ACL2 than in Coq; in our system, they are the only way to express induction schemes, while in Coq induction schemes can be constructed manually in explicit proofs.

Modular ACL2 does not permit recursive linking as in PLT Scheme units; introducing new recursion in compound modules might invalidate termination proofs. Our language also does not support nested modules as in ML structures, tagged imports and exports as in units, or a host of other possible operations on interfaces and modules. These limit the possibilities for compound modules; for instance, a module may not include two implementations of a single interface. These features represent future directions for improvement.

```
(defun setp (xs) (no-duplicatesp-equal xs))
(defun add (x xs) (add-to-set-eql x xs))

(defun add-all (xs ys)
  (cond ((endp xs) ys)
        ((consp xs) (add (car xs) (add-all (cdr xs) ys)))))

(defthm add-preserves-setp
  (implies (setp xs)
           (setp (add x xs))))

(defthm add-all-preserves-setp
  (implies (and (true-listp xs) (setp ys))
           (setp (add-all xs ys))))

(add-all (list 1 2 3) (list 2 3 4))
```

Figure 1. A finite set representation in ACL2.

3. Theorem Proving with ACL2

As far as this paper is concerned, an ACL2 program consists of a sequence of function definitions, conjecture statements, and expressions. Function definitions may be recursive, but may not have forward references. A conjecture is a named expression with free variables. Lastly, an expression applies primitive operations and previously-defined functions to atomic and compound values.

In its default "logic mode", the ACL2 theorem prover attempts to *admit* each term, verifying its soundness before adding it to the database of logical rules and proceeding to the next term. Functions must be proved terminating for all possible inputs; conjectures must be established as theorems. Expressions have no logical obligations; they are simply run.

ACL2 also has a more efficient "program mode" that ignores proof obligations and runs terms unconditionally.

Figure 1 shows a short ACL2 program defining a finite set representation (setp) and functions to add one or more elements to a set (add and add-all). The program also states conjectures that add and add-all preserve the set representation.

To admit this program, ACL2 first verifies that setp, add, and add-all terminate for all possible inputs. The proofs for the non-recursive functions setp and add are trivial. For add-all, ACL2 uses its recursive structure to construct an induction scheme, which it then proves well-founded. The theorem prover records this scheme with the definition of add-all.

ACL2 finishes the proof by checking that add-preserves-setp and add-all-preserves-setp are true for all value assignments to their free variables. It verifies add-preserves-setp based on the rules for the two built-ins: **add-to-set-eql** and **no-duplicatesp-equal**. The proof of add-all-preserves-setp demands inductive reasoning about add-all. To this end, ACL2 applies the induction scheme stored with add-all.

Finally, ACL2 runs add-all on the inputs (**list** 1 2 3) and (**list** 2 3 4). Based on the admitted theorems, we can trust the result not to duplicate 2 or 3.

Now the programmer has a working implementation of sets that may be integrated into larger programs. ACL2 provides several different tools toward this end: books, packages, encapsulation, and functional instantiation [15]. Each has its benefits, but also drawbacks:

- Books provide reusable components containing verified functions and theorems. Unfortunately, they also cause namespace clashes: all definitions are exported unless explicitly declared local. These conflicts are known to cause incompatibilities among books distributed with ACL2.
- The Common Lisp package system provides namespaces, but no scoping or abstraction mechanism [20]. Multiple books may still clash by using the same package. Packages also do not introduce a logical abstraction boundary; functions and theorems in one package are fully "visible" in another.
- Encapsulation allows "local" definitions whose names and logical rules are hidden from outside proofs. This provides scope and abstraction; however, there is no mechanism to write down an explicit specification of the exported definitions. Furthermore, local definitions cannot be run; one must sacrifice executability to gain abstraction. Finally, these abstractions cannot be built top-down; there must always be a "witness" instantiation to begin the proof.
- The "functional instantiation" mechanism can be used to connect proofs based on an encapsulation to executable code. Martín-Mateos et al. [17] demonstrate how to easily apply functional instantations to generic libraries; however, neither the instantiated theory nor its generated consequences have an explicit specification.
- The "top-down" proof style presented by Kaufmann [12] simulates specifications for abstract proofs via programming patterns. These specifications limit the rules and names exported from part of a proof. They are not reusable: multiple components with the same interface need separate specifications.

```
(interface IOne
  (sig setp (xs))
  (sig add (x xs))
  (con add-preserves-setp
    (implies (setp xs)
             (setp (add x xs)))))

(interface IMany
  (extend IOne)
  (sig add-all (xs ys))
  (con add-all-preserves-setp
    (implies (and (true-listp xs) (setp ys))
             (setp (add-all xs ys)))))

(module Many
  (import IOne)
  (defun add-all (xs ys)
    (cond ((endp xs) ys)
          ((consp xs) (add (car xs) (add-all (cdr xs) ys)))))
  (export IMany))

(module One
  (defun setp (xs) (no-duplicatesp-equal xs))
  (defun add (x xs) (add-to-set-eql x xs))
  (export IOne))

(link OneOrMany (One Many))
(invoke OneOrMany)
(add-all (list 1 2 3) (list 2 3 4))
```

Figure 2. A finite set representation in Modular ACL2.

4. Modular Reasoning in ACL2

In a recent report [5], we presented a module system for ACL2, providing a consolidated system for specification, abstraction, and the management of namespaces and components.

Our new language, dubbed Modular ACL2, introduces interfaces and modules. *Interfaces* provide abstract specifications of functions, dubbed *signatures*, and theorems, dubbed *contracts*. *Atomic modules* supply implementations for one or more interfaces, possibly based on other interfaces. *Compound modules* link together multiple modules, using the implementations of one to satisfy the assumptions of another. Modules with no further assumptions may be *invoked*; their exported functions may be called by external expressions.

The finite-set example can be rewritten as a Modular ACL2 program in which add and add-all are specified and implemented separately; see figure 2. The program starts with two interfaces. The first, IOne, specifies setp and add with *signatures* describing their name and arity. It states add-preserves-setp as a *contract* constraining the signatures above. The second interface, IMany, is an *extension* of IOne; equivalently, IOne is a *dependency* of IMany. This allows contracts in IMany to refer to signatures from IOne, and obligates implementations of IMany to include some implementation of IOne. The extension declaration is followed by a signature for add-all and the contract add-all-preserves-setp, which constrains setp (from IOne) and add-all.

Two atomic modules follow the interfaces. The module Many *imports* the interface IOne; subsequent definitions may refer to setp and add. In turn, Many defines add-all and exports IMany.

The module One *exports* the interface IOne. This supplies the functions setp and add as implementations of IOne's signatures, and obligates them to satsify the contract add-preserves-setp.

Next, the program constructs OneOrMany by *linking* together One and Many. This yields a module with One's implementations of setp and add and Many's implementation of add-all. The new module does not rely on any imports.

Finally, the program *invokes* OneOrMany, which makes setp, add, and add-all available globally. Hence, the final expression has the same meaning as in the monolithic program of figure 1.

Our model of Modular ACL2 comes with a two-pronged semantics: one side produces proof obligations for each atomic module, and the other produces an executable program from the modules invoked at the top level. A program is considered to be verified if the obligations of each atomic module can be proved in separate ACL2 sessions. The theorem prover must be restarted after each module to erase the assumptions based on its imports. The executable program comprises the definitions from all linked and invoked modules; these are run in ACL2's program mode to avoid redundant proof efforts. Soundness follows from an argument that if ACL2 verifies the atomic modules' obligations, they hold for the executable form of the program (even though ACL2 might not find their proofs in logic mode).

A few experiments with verification in Modular ACL2 demonstrate its ability to provide abstraction and reusability. In a variant of Moore's graph search case study [12], we specify the graph representation and search algorithm separately, and verify two implementations of each. In another, we verify properties of a simple video game called "Worm".

The third experiment reveals a significant drawback of the language. The experiment specifies the equivalence of two interpreters via four interfaces, shown partially in figure 3. The problem is that our interfaces can express functions (as signatures) and theorems (as contracts), but they cannot specify induction schemes.

The ILanguage interface provides a representation for expressions, recognized by the predicate expr-p. An expression may be

```
(interface ILanguage
  (sig expr-p (x))
  ... more predicates, constructors, and selectors ...
  (con expr/calc
    (iff (and (op-p o) (expr-p l) (expr-p r))
         (expr-p (calc o l r))))
  (con expr/integer
    (iff (and (expr-p e) (not (calc-p e)))
         (integerp e)))
  ... more contracts about expr-p, calc-p, and op-p... )

(interface ISmallStep
  (extend ILanguage)
  (sig single-step (e))
  (con single-step-plus
    (implies (and (integerp l) (integerp r))
             (equal (single-step (calc '+ l r)) (+ l r))))
  ... more contracts about single-step...
  (sig step-all (e))
  (con step-all-calc
    (implies (calc-p e)
             (equal (step-all e) (step-all (single-step e)))))
  ... more contracts about step-all... )

(interface IBigStep
  (extend ILanguage)
  (sig evaluate (e))
  (con evaluate-plus
    (equal (evaluate (calc '+ l r))
           (+ (evaluate l) (evaluate r))))
  ... more contracts about evaluate... )

(interface IEquivalence
  (extend ILanguage ISmallStep IBigStep)
  (con step-all=evaluate
    (implies (expr-p e)
             (equal (step-all e) (evaluate e)))))
```

Figure 3. Excerpts from interfaces in the interpreter experiment.

an integer or a "calculation" recognized by calc-p. A calculation applies an operator (recognized by op-p) to left and right operands.

A reduction semantics for the language is specified by ISmallStep. It describes a single-step function on expressions that reduces one calculation on integers at a time and a step-all function that performs single-step until no calculations remain.

We describe recursive evaluation in IBigStep, extending ILanguage with a function that yields an integer for each expression.

In IEquivalence, we extend ILanguage, ISmallStep, and IBigStep. Then we state the claim that step-all and evaluate produce the same result when given an expression satisfying expr-p. The module system guarantees that the implementation of an interface shares the implementation of its (transitive) dependencies, so we may rely on step-all, evaluate, and the step-all=evaluate contract to use the same definition of expr-p.

These interfaces implicitly introduce two patterns of recursion: traversing an expression and reducing an expression to a value. The proof of step-all=evaluate must reason inductively about both. ACL2 allows various annotations on conjectures that choose from or combine known induction schemes; however, a new scheme can only be introduced by a complete function definition. Because the relevant function definitions are in other modules, we found it necessary to duplicate the expr-p and step-all functions in the module implementing IEquivalence, prove them equal to the imported version, and reason from the duplicates.

In general, this problem arises whenever a program introduces a pattern of recursion, whether it is a data structure (as with expr-p) or an algorithm (as with step-all), and the function definition lies in a different module from a proof that uses it. Our other experiments use lists and list traversal for all their inductive definitions. ACL2 can therefore use its built-in induction schemes regardless of module boundaries. Proofs using other data structures or non-structural recursion (e.g., quicksort) must introduce new induction schemes to support reasoning across module boundaries.

5. Manifest Functions: Induction Across Boundaries

This section presents our revision of Modular ACL2. It starts with a description of the design space for manifest functions, followed by two separate semantics: one for verification and one for execution.

5.1 Language Design

To express induction schemes for abstraction boundaries, we introduce *manifest functions* into interfaces. In addition to signatures, contracts, and dependencies, interfaces may now express functions with a name, argument list, and body expression that may refer to other manifest functions and opaque function signatures. These specifications supply an exporting module with a function definition that must be proved terminating, and allow an importing module to use the resulting logical rules: the body of the function and its attending induction scheme, if any. Any opaque signatures to which the manifest function refers remain abstract. Thus, interfaces as a whole are *translucent*, analogous to the ML signatures of Harper and Lillibridge [10].

The design of manifest functions is motivated by ACL2's method of inferring induction schemes from functions. A manifest function provides exactly the definition ACL2 needs for inference. An alternate design might allow users to specify induction schemes abstractly. The verification process for Modular ACL2 would still have to synthesize a function definition to communicate the scheme to the theorem prover. Manifest functions avoid this extra step, thus simplifying the correlation between Modular ACL2 code and verified ACL2 code. Users wishing to separate induction schemes from program behavior can export "dummy" manifest functions with appropriate recursive structure but trivial output, e.g., returning **nil** in all clauses. The resulting induction scheme can be used in other modules to reason about other functions, even abstract ones introduced by signatures.

The new grammar of Modular ACL2 is shown in figure 5. It extends the core grammar of ACL2 in figure 4. Keywords are set in **bold** and nonterminals in *italics*. We write $\overrightarrow{X}$ to denote a sequence of terms of the form X or a set when order is insignificant. A sequence of length n is written $\overrightarrow{X}^n$.

$$\begin{array}{lcl}
prog & = & \overrightarrow{term} \\
term & = & defn \mid expr \\
defn & = & dfun \mid dthm \mid dstub \mid dskip \\
dfun & = & (\textbf{defun}\ f\ (\overrightarrow{x})\ expr) \\
dthm & = & (\textbf{defthm}\ f\ expr) \\
dstub & = & (\textbf{defstub}\ f\ (\overrightarrow{x})\ \textbf{t}) \\
dskip & = & (\textbf{skip-proofs}\ defn)
\end{array}$$

Figure 4. The core grammar of ACL2.

ACL2 programs consist of a sequence of definitions and expressions. Definitions may be functions, conjectures, or stubs, which provide a function name and arity but no implementation. Definitions may be wrapped in **skip-proofs**, which informs the theorem prover to admit them without proof.[1] ACL2 includes two variable namespaces: one for functions and conjectures (f) and another for function parameters and local variables (x).

$$\begin{array}{lcl}
mprog & = & \overrightarrow{comp} \\
comp & = & ifc \mid mod \mid link \mid inv \mid expr \\
ifc & = & (\textbf{interface}\ n\ \overrightarrow{spec}) \\
mod & = & (\textbf{module}\ n\ \overrightarrow{body}) \\
link & = & (\textbf{link}\ n\ (n\ n)) \\
inv & = & (\textbf{invoke}\ n) \\
spec & = & fun \mid sig \mid con \mid ext \\
fun & = & (\textbf{fun}\ f\ (\overrightarrow{x})\ expr) \\
sig & = & (\textbf{sig}\ f\ (\overrightarrow{x})) \\
con & = & (\textbf{con}\ f\ expr) \\
ext & = & (\textbf{extend}\ n) \\
body & = & im \mid ex \mid defn \\
im & = & (\textbf{import}\ n\ \overrightarrow{re}) \\
ex & = & (\textbf{export}\ n\ \overrightarrow{re}) \\
re & = & (f\ f)
\end{array}$$

Figure 5. The grammar of Modular ACL2.

Modular ACL2 programs ($mprog$) consist of a sequence of components. A component may be an interface, atomic module, compound module, module invocation, or top level expression. Interfaces and modules come with names; an interface contains a sequence of specifications; an atomic module contains a sequence of body terms; and a compound module is constructed based on the names of two constituent modules.

An interface may specify manifest functions, opaque signatures, contracts, or dependencies. A manifest function exposes the actual implementation of a function, including a name, argument list, and body expression. An opaque signature provides only a name and argument list. A contract has a name and a logical claim. Other interfaces may be extended by name, thus introducing a dependency.

The body of a module may include definitions, imports, and exports. Imports and exports name an interface and provide a sequence of renamings that map function names in the interface to function names inside the module. Imports provide a set of specifications that the module may rely on; exports describe a set of specifications that the module satisfies. Since manifest functions are defined in interfaces, an exporting module need not define them internally; the export clause implicitly defines the function, and subsequent definitions in the module may refer to it.

Compound modules are linked *nominally* in Modular ACL2. Any names joined between two modules by linking must be imported and exported via the same interface. This ensures the "consumer" module assumes precisely those contracts about its imports that the "producer" module ensures.

For the purposes of this paper, we put further syntactic restrictions on Modular ACL2 programs. An interface must explicitly extend all its transitive dependencies. A module must explicitly import or export all transitive dependencies of its imports and exports. Each import and export must provide explicit internal names for all functions and theorems from the relevant interface. These restrictions simplify verification and compilation, but complicate programming. We therefore assume a surface syntax without these restrictions and an elaboration process which synthesizes the implicit

```
(interface ILanguage
  ...signatures except for expr-p...
  (fun expr-p (v)
    (cond ((integerp v) t)
          ((calc-p v) (and (op-p (calc-op v))
                           (expr-p (calc-left v))
                           (expr-p (calc-right v))))))
  ...contracts about calc-p and op-p...)

(interface ISmallStep
  (extend ILanguage)
  (sig single-step (e))
  (con single-step-plus
    (implies (and (integerp l) (integerp r))
             (equal (single-step (calc '+ l r)) (+ l r))))
  ...more contracts about single-step...
  (fun step-all (e)
    (cond ((integerp e) e)
          ((calc-p e) (step-all (single-step e))))))
```

Figure 6. Excerpts from modified interpreter interfaces.

dependencies, imports, exports, and names, though for brevity's sake we do not present them.

In this system, we can reformulate the interpreter example from the preceding section (see figure 3) with manifest functions. Figure 6 shows the modified portions of the interfaces. In ILanguage, the signature and contracts for expr-p are replaced by a manifest function definition. This definition adds to the previous version an induction scheme for traversing an expression through the operands of a calculation. A module exporting this new interface must ensure that the calc-left and calc-right of a calculation are smaller than the original expression, and a module importing it may reason with the new induction scheme.

Similarly, we replace the step-all signature and related contracts in ISmallStep with a manifest function definition. This establishes an induction scheme for reducing a calculation step-by-step to an integer. An exporting module must ensure that step-all terminates, i.e., that single-step brings an expression closer to a final result. Importing modules may then reason about the (finite) reduction sequence of an expression.

Now the proof of step-all=evaluate completes immediatly. It uses the manifest function definitions of step-all and expr-p to reason inductively about step-all and evaluate, respectively.

5.2 Verifying Modules

Programs in Modular ACL2 are verified by extracting proof obligations for each atomic module and submitting them to a new session of the ACL2 theorem prover. If ACL2 admits each set of proof obligations, the entire modular program is verified, including the compound modules. We formalize this process in figures 7 and 8.

For the verification semantics, we introduce two kinds of environments: interface environments (Γ_i) and renaming environments (Γ_r), represented as sequences whose elements may be looked up by name, i.e., the first f appearing syntactically. Figure 8 defines substitution on (Modular) ACL2 terms.

The main judgment in the verification of modular programs is $\vdash_p mprog$, meaning that the program's components can be verified by ACL2. This is defined in terms of $\Gamma_i \vdash_c \overrightarrow{comp}$, meaning that the components can be verified in the context of additional interfaces, and $\vdash_s prog$, meaning that ACL2 verifies the program. We defer the definition of this judgment until section 6.

[1] The **skip-proofs** form may admit unsound conjectures, and is usually reserved for intermediate stages of proof development. See section 5.2.

PFMPROG
$$\frac{\epsilon \vdash_c \mathit{mprog}}{\vdash_p \mathit{mprog}}$$

PFCOMP0
$$\frac{}{\Gamma_i \vdash_c \epsilon}$$

PFCEXPR
$$\frac{\Gamma_i \vdash_c \overrightarrow{\mathit{comp}}}{\Gamma_i \vdash_c \mathit{expr}\ \overrightarrow{\mathit{comp}}}$$

PFIFC
$$\frac{\Gamma_i\ \mathit{ifc} \vdash_c \overrightarrow{\mathit{comp}}}{\Gamma_i \vdash_c \mathit{ifc}\ \overrightarrow{\mathit{comp}}}$$

PFMOD
$$\frac{\vdash_s \textsf{\textit{obligations}}(\Gamma_i, \mathit{mod}) \qquad \Gamma_i \vdash_c \overrightarrow{\mathit{comp}}}{\Gamma_i \vdash_c \mathit{mod}\ \overrightarrow{\mathit{comp}}}$$

PFLINK
$$\frac{\Gamma_i \vdash_c \overrightarrow{\mathit{comp}}}{\Gamma_i \vdash_c \mathit{link}\ \overrightarrow{\mathit{comp}}}$$

PFINV
$$\frac{\Gamma_i \vdash_c \overrightarrow{\mathit{comp}}}{\Gamma_i \vdash_c \mathit{inv}\ \overrightarrow{\mathit{comp}}}$$

Figure 7. Inference rules for Modular ACL2 verification.

$$\begin{aligned}
&\textsf{\textit{obligations}} : \Gamma_i, \mathit{mod} \rightarrow \mathit{prog}\\
&\quad \textsf{\textit{obligations}}(\Gamma_i, (\textbf{module}\ n\ \overrightarrow{\mathit{body}}))\\
&\quad = \overrightarrow{\textsf{\textit{verify}}(\Gamma_i, \mathit{body})}
\end{aligned}$$

$$\begin{aligned}
&\textsf{\textit{verify}} : \Gamma_i, \mathit{body} \rightarrow \overrightarrow{\mathit{defn}}\\
&\quad \textsf{\textit{verify}}(\Gamma_i, \mathit{defn}) && = \mathit{defn}\\
&\quad \textsf{\textit{verify}}(\Gamma_i, (\textbf{import}\ n\ \overrightarrow{(f_1\ f_2)})) && = \overrightarrow{\textsf{\textit{assume}}(\mathit{spec})[\overrightarrow{f_1 = f_2}]} \text{ where } \Gamma_i(n) = (\textbf{interface}\ n\ \overrightarrow{\mathit{spec}})\\
&\quad \textsf{\textit{verify}}(\Gamma_i, (\textbf{export}\ n\ \overrightarrow{(f_1\ f_2)})) && = \textsf{\textit{assert}}(\mathit{spec})[\overrightarrow{f_1 = f_2}] \text{ where } \Gamma_i(n) = (\textbf{interface}\ n\ \overrightarrow{\mathit{spec}})
\end{aligned}$$

$$\begin{aligned}
&\textsf{\textit{assume}} : \mathit{spec} \rightarrow \overrightarrow{\mathit{term}}\\
&\quad \textsf{\textit{assume}}((\textbf{fun}\ f\ (\overrightarrow{x})\ \mathit{expr})) && = (\textbf{skip-proofs}\ (\textbf{defun}\ f\ (\overrightarrow{x})\ \mathit{expr}))\\
&\quad \textsf{\textit{assume}}((\textbf{sig}\ f\ (\overrightarrow{x}))) && = (\textbf{defstub}\ f\ (\overrightarrow{x})\ \textbf{t})\\
&\quad \textsf{\textit{assume}}((\textbf{con}\ f\ \mathit{expr})) && = (\textbf{skip-proofs}\ (\textbf{defthm}\ f\ \mathit{expr}))\\
&\quad \textsf{\textit{assume}}((\textbf{extend}\ n)) && = \epsilon
\end{aligned}$$

$$\begin{aligned}
&\textsf{\textit{assert}} : \mathit{spec} \rightarrow \overrightarrow{\mathit{term}}\\
&\quad \textsf{\textit{assert}}((\textbf{fun}\ f\ (\overrightarrow{x})\ \mathit{expr})) && = (\textbf{defun}\ f\ (\overrightarrow{x})\ \mathit{expr})\\
&\quad \textsf{\textit{assert}}((\textbf{sig}\ f\ (\overrightarrow{x}))) && = \epsilon\\
&\quad \textsf{\textit{assert}}((\textbf{con}\ f\ \mathit{expr})) && = (\textbf{defthm}\ f\ \mathit{expr})\\
&\quad \textsf{\textit{assert}}((\textbf{extend}\ n)) && = \epsilon
\end{aligned}$$

$$\begin{aligned}
&\cdot[\cdot = \cdot] : \mathit{body}, f, f \rightarrow \mathit{body}\\
&\quad (\textbf{defun}\ f\ (\overrightarrow{x})\ \mathit{expr})[f_1 = f_2] && = (\textbf{defun}\ f[f_1 = f_2]\ (\overrightarrow{x})\ \mathit{expr}[f_1 = f_2])\\
&\quad (\textbf{defthm}\ f\ \mathit{expr})[f_1 = f_2] && = (\textbf{defthm}\ f[f_1 = f_2]\ \mathit{expr}[f_1 = f_2])\\
&\quad (\textbf{defstub}\ f\ (\overrightarrow{x})\ \textbf{t})[f_1 = f_2] && = (\textbf{defstub}\ f[f_1 = f_2]\ (\overrightarrow{x})\ \textbf{t})\\
&\quad (\textbf{skip-proofs}\ \mathit{defn})[f_1 = f_2] && = (\textbf{skip-proofs}\ \mathit{defn}[f_1 = f_2])\\
&\quad (\textbf{import}\ n\ \overrightarrow{(f_3\ f_4)})[f_1 = f_2] && = (\textbf{import}\ n\ \overrightarrow{(f_3\ f_4[f_1 = f_2])})\\
&\quad (\textbf{export}\ n\ \overrightarrow{(f_3\ f_4)})[f_1 = f_2] && = (\textbf{export}\ n\ \overrightarrow{(f_3\ f_4[f_1 = f_2])})
\end{aligned}$$

Figure 8. Metafunctions for Modular ACL2 verification.

Atomic modules entail proof obligations that must be verified by ACL2, constructed by the *obligations* metafunction. Compound modules entail the proof obligations of their combined exports, given the assumption of their combined imports except those resolved by linking. These obligations are fulfilled by their components when verified separately. Each constituent module entails a proof of its own exports; nominal interface linking ensures that the "producer" module's obligations include precisely those assumptions of the "consumer" module that are discharged by linking. Thus, compound modules do not contribute proof obligations beyond those of their constituents. Interfaces only generate proof obligations insofar as they contribute to atomic modules that import or export them; module invocations and top-level expressions are for execution only and do not generate proof obligations at all.

Each term in a module's body is translated to ACL2 definitions representing its logical meaning by the *verify* metafunction. An import becomes an assumption of a correct implementation of the named interface, constructed by the *assume* metafunction. Signatures are represented as stubs; manifest functions and contracts are represented as function and conjecture definitions wrapped in **skip-proofs**. An export becomes a claim to be verified, constructed by the *assert* metafunction. Manifest functions and contracts map to function and conjecture definitions. In both imports and exports, an **extend** clause requires the enclosing module to import or export the extended interface as well. The **extend** clause inserts no definitions itself; the extended interface is instead translated separately.

5.3 Verification Example

To illustrate the verification process, we construct proof obligations for the set representation from figure 2. To verify this program, we must establish the correctness of two modules: Many and One. We ignore for this example the syntactic restriction that all imports and exports require explicit renaming.

First we verify Many. The *assume* metafunction converts the specifications of IOne into stubs for the imported signatures and an assumption that they satisfy add-preserves-setp. Then we include add-all and finally use *assert* to construct a conjecture that the definitions satisfy add-all-preserves-setp.

```
(defstub setp (xs) t)
(defstub add (x xs) t)
(skip-proofs
 (defthm add-preserves-setp
   (implies (setp xs)
            (setp (add x xs)))))

(defun add-all (xs ys)
  (cond ((endp xs) ys)
        ((consp xs) (add (car xs) (add-all (cdr xs) ys)))))

(defthm add-all-preserves-setp
  (implies (and (true-listp xs) (setp ys))
           (setp (add-all xs ys))))
```

The last two definitions are directly from figure 1, while the first three are translations of the imported interface. This permits abstract reasoning about IOne within Many, as the theorem prover doesn't have implementations for setp or add.

Next we construct proof obligations for One. By the definitions of *obligations* and *verify*, we concatenate its internal definitions of setp and add with an assertion that add-all-preserves-setp holds. We apply the *assert* metafunction to construct the final definitions:

```
(defun setp (xs) (no-duplicatesp-equal xs))
(defun add (x xs) (add-to-set-eql x xs))

(defthm add-preserves-setp
  (implies (setp xs)
           (setp (add x xs))))
```

$$
\begin{aligned}
&\mathit{execute} : mprog \rightarrow prog\\
&\quad \mathit{execute}(mprog) && = \mathit{compile}(\epsilon, \epsilon, \epsilon, mprog)\\
\\
&\mathit{compile} : \Gamma_i, \Gamma_m, \Gamma_r, \overrightarrow{comp} \rightarrow \overrightarrow{term}\\
&\quad \mathit{compile}(\Gamma_i, \Gamma_m, \Gamma_r, ifc\ \overrightarrow{comp}) && = \mathit{compile}(\Gamma_i\ ifc, \Gamma_m, \Gamma_r, \overrightarrow{comp})\\
&\quad \mathit{compile}(\Gamma_i, \Gamma_m, \Gamma_r, mod\ \overrightarrow{comp}) && = \mathit{compile}(\Gamma_i, \Gamma_m\ mod, \Gamma_r, \overrightarrow{comp})\\
&\quad \mathit{compile}(\Gamma_i, \Gamma_m, \Gamma_r, (\textbf{link}\ n\ (n_1\ n_2))\ \overrightarrow{comp}) && = \mathit{compile}(\Gamma_i, \Gamma_m\ \mathit{link}(\Gamma_i, n, \Gamma_m(n_1), \Gamma_m(n_2)), \Gamma_r, \overrightarrow{comp})\\
&\quad \mathit{compile}(\Gamma_i, \Gamma_m, \Gamma_r, (\textbf{invoke}\ n)\ \overrightarrow{comp}) && = \overrightarrow{\mathit{verify}(\Gamma_i, body_2)}\ \mathit{compile}(\Gamma_i, \Gamma_m, \Gamma_r\ \overrightarrow{re}, \overrightarrow{comp})\\
&&& \quad \text{where } \Gamma_m(n) = (\textbf{module}\ n\ \overrightarrow{body_1})\\
&&& \quad \text{and } \mathit{rename}(\overrightarrow{body_1}) = \overrightarrow{body_2}\\
&&& \quad \text{and } \{\overrightarrow{re_0} \mid (\textbf{import}\ n\ \overrightarrow{re_0}) \in \overrightarrow{body_2} \text{ or } (\textbf{export}\ n\ \overrightarrow{re_0}) \in \overrightarrow{body_2}\} = \overrightarrow{re}\\
&\quad \mathit{compile}(\Gamma_i, \Gamma_m, \overrightarrow{(f_1\ f_2)}, expr\ \overrightarrow{comp}) && = expr[\overrightarrow{f_1 = f_2}]\ \mathit{compile}(\Gamma_i, \Gamma_m, \overrightarrow{(f_1\ f_2)}, \overrightarrow{comp})\\
&\quad \mathit{compile}(\Gamma_i, \Gamma_m, \Gamma_r, \epsilon) && = \epsilon\\
\\
&\mathit{link} : \Gamma_i, n, mod, mod \rightarrow mod\\
&\quad \mathit{link}(\Gamma_i, n, (\textbf{module}\ n_1\ \overrightarrow{body_1}), && = (\textbf{module}\ n\ \overrightarrow{body_3}\ \overrightarrow{body_4})\\
&\qquad\qquad (\textbf{module}\ n_2\ \overrightarrow{body_2})) && \quad \text{where } \mathit{rename}(\overrightarrow{body_1}) = \overrightarrow{body_3}\\
&&& \quad \text{and } \mathit{resolve}(\overrightarrow{body_3}, \mathit{rename}(\overrightarrow{body_2})) = \overrightarrow{body_4}\\
\\
&\mathit{resolve} : \overrightarrow{body}, \overrightarrow{body} \rightarrow \overrightarrow{body}\\
&\quad \mathit{resolve}(\overrightarrow{body_1}, \epsilon) && = \epsilon\\
&\quad \mathit{resolve}(\overrightarrow{body_1}, (\textbf{import}\ n\ \overrightarrow{(f\ f_1)})\ \overrightarrow{body_2}) && = \mathit{resolve}(\overrightarrow{body_1}, \overrightarrow{body_2[\overrightarrow{f_1 = f_2}]}) \text{ if } (\textbf{import}\ n\ \overrightarrow{(f\ f_2)}) \in \overrightarrow{body_1}\\
&\quad \mathit{resolve}(\overrightarrow{body_1}, (\textbf{import}\ n\ \overrightarrow{(f\ f_1)})\ \overrightarrow{body_2}) && = \mathit{resolve}(\overrightarrow{body_1}, \overrightarrow{body_2[\overrightarrow{f_1 = f_2}]}) \text{ if } (\textbf{export}\ n\ \overrightarrow{(f\ f_2)}) \in \overrightarrow{body_1}\\
&\quad \mathit{resolve}(\overrightarrow{body_1}, (\textbf{import}\ n\ \overrightarrow{re})\ \overrightarrow{body_2}) && = (\textbf{import}\ n\ \overrightarrow{re})\ \mathit{resolve}(\overrightarrow{body_1}, \overrightarrow{body_2}) \text{ if } n \notin \overrightarrow{body_1}\\
&\quad \mathit{resolve}(\overrightarrow{body_1}, defn\ \overrightarrow{body_2}) && = defn\ \mathit{resolve}(\overrightarrow{body_1}, \overrightarrow{body_2})\\
&\quad \mathit{resolve}(\overrightarrow{body_1}, ex\ \overrightarrow{body_2}) && = ex\ \mathit{resolve}(\overrightarrow{body_1}, \overrightarrow{body_2})\\
\\
&\mathit{rename} : \overrightarrow{body} \rightarrow \overrightarrow{body}\\
&\quad \mathit{rename}(\overrightarrow{body}) && = \overrightarrow{body[\overrightarrow{f_1 = f_2}]} \text{ where } \overrightarrow{\mathit{introduced}(body)} = \overrightarrow{f_1}^n \text{ and } \overrightarrow{f_2}^n \text{ fresh}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{introduced} : body \rightarrow \overrightarrow{f}\\
&\quad \mathit{introduced}((\textbf{import}\ n\ \overrightarrow{(f_1\ f_2)})) = \overrightarrow{f_2} && \mathit{introduced}((\textbf{defstub}\ f\ (\overrightarrow{x})\ \textbf{t})) = f\\
&\quad \mathit{introduced}(ex) = \epsilon && \mathit{introduced}((\textbf{defthm}\ f\ expr)) = f\\
&\quad \mathit{introduced}((\textbf{defun}\ f\ (\overrightarrow{x})\ expr)) = f && \mathit{introduced}((\textbf{skip-proofs}\ defn)) = \mathit{introduced}(defn)
\end{aligned}
$$

Figure 9. Translation from a Modular ACL2 program to an executable ACL2 program.

These definitions are all present in the original set representation of figure 1; modules without imports represent concrete reasoning.

The compound module OneOrMany links Many to One. It shares both their exports; since both are verified, so are the exports of OneOrMany. Note that One provides implementations of setp and add and a proof of add-all-preserves-setp. The verification of OneOrMany relies on these verified definitions in place of the unverified **defstub** and **skip-proofs** forms from Many's proof obligation. This substitution of verified definitions for assumptions is the basis of our soundness theorem; it roughly corresponds to the discharge of an implication.

5.4 Executing Modules

The execution of Modular ACL2 programs is defined by the metafunction *execute*, shown in figure 9, which transforms a Modular ACL2 program to an ACL2 program. We introduce sequences of modules as environments (Γ_m) for use in compilation. During compilation, we maintain environments of interfaces, modules, and renamings, which map top level function names to implementations provided by invoked modules. The compilation process adds interfaces, atomic modules, and compound modules to the appropriate environments. The constituents of compound modules are extracted from the environment and linked first.

Turning compound modules into atomic modules is the key step in compilation. The metafunction *link* applies *rename* to the body of both constituent modules, giving their definitions fresh names to prevent name clashes. It then links imports of the second module ("consumer") to exports of the first ("producer") via the *resolve* metafunction. The same process coalesces shared imports. Linking is one-directional—exports flow from the producer to the consumer—to prevent introducing new recursion that might invalidate termination proofs. The resulting module contains the definitions, exports, and unresolved imports of both constituents.

The *resolve* metafunction consumes terms from the body of producer and consumer modules and processes each term from the consumer in order. If it reaches an import that coincides with an import or export of the producer, it substitutes the internal names from the producer, drops the import, and continues. Otherwise, terms from the consumer module are left unchanged.

We extract executable definitions from invoked modules in the same way we extract proof obligations during verification (*verify*). This produces each module's internal definitions, along with the contracts and manifest functions of their exported interfaces (as

$$\text{PFPROG}\quad \frac{\Gamma_e^0 \vdash_t \mathit{prog}}{\vdash_s \mathit{prog}} \qquad \text{PFTERM0}\quad \frac{}{\Gamma_e \vdash_t \epsilon} \qquad \text{PFTEXPR}\quad \frac{\Gamma_e \vdash_t \overrightarrow{\mathit{term}}}{\Gamma_e \vdash_t \mathit{expr}\ \overrightarrow{\mathit{term}}} \qquad \text{PFTDEFN}\quad \frac{\Gamma_e \vdash_d \mathit{defn} \qquad \Gamma_e\ \mathsf{theory}(\mathit{defn}) \vdash_t \overrightarrow{\mathit{term}}}{\Gamma_e \vdash_t \mathit{defn}\ \overrightarrow{\mathit{term}}}$$

$$\text{PFFUN}\quad \frac{\Gamma_e \vdash_e \mathsf{measure}(f, \overrightarrow{x}, \mathit{expr})}{\Gamma_e \vdash_d (\textbf{defun}\ f\ (\overrightarrow{x})\ \mathit{expr})} \qquad \text{PFTHM}\quad \frac{\Gamma_e \vdash_e \mathit{expr}}{\Gamma_e \vdash_d (\textbf{defthm}\ f\ \mathit{expr})} \qquad \text{PFSTUB}\quad \frac{}{\Gamma_e \vdash_d \mathit{dstub}} \qquad \text{PFSKIP}\quad \frac{}{\Gamma_e \vdash_d \mathit{dskip}}$$

Figure 10. Inference rules for ACL2 verification.

$$\begin{aligned}
&\mathsf{theory} : \mathit{defn} \to \overrightarrow{\mathit{expr}}\\
&\quad \mathsf{theory}((\textbf{defun}\ f\ (\overrightarrow{x})\ \mathit{expr})) = (\textbf{equal}\ (f\ \overrightarrow{x})\ \mathit{expr})\ \mathsf{induction}(f, \overrightarrow{x}, \mathit{expr})\\
&\quad \mathsf{theory}((\textbf{defthm}\ f\ \mathit{expr})) \quad = \mathit{expr}\\
&\quad \mathsf{theory}((\textbf{defstub}\ f\ (\overrightarrow{x})\ \textbf{t})) \quad = \epsilon\\
&\quad \mathsf{theory}((\textbf{skip-proofs}\ \mathit{defn})) \ = \mathsf{theory}(\mathit{defn})
\end{aligned}$$

$$\begin{aligned}
&\mathsf{measure} : f, \overrightarrow{x}, \mathit{expr} \to \mathit{expr}\\
&\quad \text{termination conditions (omitted)}\\
&\mathsf{induction} : f, \overrightarrow{x}, \mathit{expr} \to \mathit{expr}\\
&\quad \text{induction schemes (omitted)}
\end{aligned}$$

Figure 11. Metafunctions for ACL2 verification.

defthm and **defun** forms). Modules with unresolved imports may not be invoked, so the process does not generate any abstract definitions (**defstub** or **skip-proofs**). The extracted definitions are given fresh names to prevent clashes, and the renaming environment is updated.

Top level expressions are linked to invoked modules by appropriate renaming. They are then added to the executable program.

5.5 Execution Example

Once again, consider the finite set representation from figure 2. To construct executable code for this program, we must first link together the atomic modules One and Many into OneOrMany.

The linked, atomic form of OneOrMany constructed by *link* and *resolve* contains the definitions and exports of One and Many:

```
(module OneOrMany
  (defun setp (xs) (no-duplicatesp-equal xs))
  (defun add (x xs) (add-to-set-eql x xs))
  (export IOne)
  (defun add-all (xs ys)
    (cond ((endp xs) ys)
          ((consp xs) (add (car xs) (add-all (cdr xs) ys)))))
  (export IMany))
```

It no longer contains the import of IOne from Many; add-all and add-all-preserves-setp now refer to the concrete definitions of setp and add from One.

Compilation completes by invoking OneOrMany, exposing its definitions and the assertions of its exported contracts and allowing top-level expressions to refer to them:

```
(defun setp (xs) (no-duplicatesp-equal xs))
(defun add (x xs) (add-to-set-eql x xs))

(defthm add-preserves-setp
  (implies (setp xs)
           (setp (add x xs))))

(defun add-all (xs ys)
  (cond ((endp xs) ys)
        ((consp xs) (add (car xs) (add-all (cdr xs) ys)))))

(defthm add-all-preserves-setp
  (implies (and (true-listp xs) (setp ys))
           (setp (add-all xs ys))))

(add-all (list 1 2 3) (list 2 3 4))
```

Aside from the reordering of add-preserves-setp and add-all, this program is the same as the original monolithic program from figure 1. Modular ACL2's compilation process has produced a program that contains assertions of all contracts exported by the invoked module OneOrMany, and whose soundness follows from the verification of the atomic modules One and Many. Due to the verification step, this program can be safely run in ACL2's program mode, bypassing logical verification for efficient execution.

6. Soundness and Expressivity

Adding linguistic machinery to the programming language of a theorem prover demands a rigorous soundness proof. Our previous report [5] skipped this step in favor of experiments concerning the pragmatics of our modules. In this section we supply a complete soundness theorem. We establish that the translation to executable code preserves verified contracts. Therefore, once a program's atomic modules have been verified, its fully-linked executable form is verified as well. The proof not only demonstrates the correctness of our approach, but guarantees modular reasoning; conclusions drawn about a module once can be applied anywhere it may be linked.

We also establish the expressivity of our system. As our initial experiments demonstrated, Modular ACL2 without manifest functions could not express all possible decompositions of a proof into modules. We present a proof that our new system can, thus confirming the completeness of our specification language with respect to the theorem prover's logical rules.

The section starts with a formal model of the ACL2 logic. This is followed by our soundness proof. It finishes with a proof of the expressivity of Modular ACL2 by inspection of the language.

6.1 The Logical Meaning of ACL2

Before we can establish the soundness of Modular ACL2, we must describe what it means for an ACL2 program to be provable. Our formalization, shown in figure 10, is based on Kaufmann and Moore's work [14, 15]. Supporting definitions are in figure 11.

The primary judgment, $\vdash_s\ \mathit{prog}$, defines the provability of whole programs. It is built up by iteration over terms. Analogously, the judgment $\Gamma_e \vdash_t \overrightarrow{\mathit{term}}$ describes the provability of terms in a "theory environment" Γ_e of expressions representing proved theorems. The environment Γ_e^0 represents the initial theory of ACL2.

Top level expressions add nothing to the environment. Definitions are verified according to the judgment $\Gamma_e \vdash_d\ \mathit{defn}$. Then their conclusions are added to the environment. The judgment $\Gamma_e \vdash_e \mathit{expr}$ means that expr is provably valid and is used to check

$$\begin{array}{lcl}
\Gamma_e \vdash_t \textsf{verify}(\Gamma_i, (\textbf{import}\ n\ \overrightarrow{(f\ f_1)})\ \overrightarrow{body}) & \equiv & \text{[def. }\textsf{verify}\text{]} \\
\Gamma_e \vdash_t \overrightarrow{\textsf{assume}(spec)[\overrightarrow{f=f_1}]}\ \textsf{verify}(\Gamma_i, \overrightarrow{body}) & \Rightarrow & \text{[lemma 7]} \\
\Gamma_e \vdash_t \overrightarrow{\textsf{assume}(spec)[\overrightarrow{f=f_2}]}\textsf{verify}(\Gamma_i, \overrightarrow{body})[\overrightarrow{f_1=f_2}] & \Rightarrow & \text{[lemma 8]} \\
\Gamma_e\ \textsf{theory}(\overrightarrow{\textsf{assume}(spec)[\overrightarrow{f=f_2}]}) \vdash_t \textsf{verify}(\Gamma_i, \overrightarrow{body})[\overrightarrow{f_1=f_2}] & \equiv & \text{[lemma 5]} \\
\Gamma_e\ \textsf{theory}(\overrightarrow{\textsf{assume}(spec)[\overrightarrow{f=f_2}]}) \vdash_t \textsf{verify}(\Gamma_i, \overrightarrow{body}[\overrightarrow{f_1=f_2}]) & \equiv & \text{[lemma 9]} \\
\Gamma_e\ \textsf{theory}(\overrightarrow{\textsf{assert}(spec)[\overrightarrow{f=f_2}]}) \vdash_t \textsf{verify}(\Gamma_i, \overrightarrow{body}[\overrightarrow{f_1=f_2}]) & \Rightarrow & \text{[ind. hyp.]} \\
\Gamma_e\ \textsf{theory}(\overrightarrow{\textsf{assert}(spec)[\overrightarrow{f=f_2}]})\ \textsf{theory}(\textsf{verify}(\Gamma_i, \overrightarrow{body_1})) \vdash_t \textsf{verify}(\Gamma_i, \textsf{resolve}(\overrightarrow{body_1}, \overrightarrow{body}[\overrightarrow{f_1=f_2}])) & \equiv & \text{[def. }\textsf{verify}\text{]} \\
\Gamma_e\ \textsf{theory}(\textsf{verify}(\Gamma_i, \overrightarrow{body_1})) \vdash_t \textsf{verify}(\Gamma_i, \textsf{resolve}(\overrightarrow{body_1}, \overrightarrow{body}[\overrightarrow{f_1=f_2}])) & \equiv & \text{[def. }\textsf{resolve}\text{]} \\
\Gamma_e\ \textsf{theory}(\textsf{verify}(\Gamma_i, \overrightarrow{body_1})) \vdash_t \textsf{verify}(\Gamma_i, \textsf{resolve}(\overrightarrow{body_1}, (\textbf{import}\ n\ \overrightarrow{(f\ f_1)})\ \overrightarrow{body})) & &
\end{array}$$

Figure 12. Equational reasoning used in lemma 3.

both explicit conjectures and the measure conjectures (termination arguments) of functions.

Function definitions require a termination argument to be verified. Termination conditions are computed by the *measure* metafunction (not shown here; see Kaufmann and Moore [14] for the details of ACL2 measure conjectures). Each admitted function contributes two expressions to the program's theory: its definition (expressed with the **equal** function) and its implicit induction scheme, if any (computed by the *induction* metafunction, also omitted).

Conjecture definitions require their body expression to be verified. Then the theorems are added to the theory. Logically this is unnecessary, as the theory already entails the conclusion; however, the addition is sound and models the rules added by the ACL2 theorem prover to aid its proof search algorithm.

Stubs have neither proof obligations nor new introduced theorems; they only introduce a name. Definitions inside **skip-proofs** are considered "provable" without verification, but introduce expressions to the theory regardless. We omit the definition of expression provability, relying instead on the prior formalization [14] and the well-known properties of first-order logic.

6.2 Proof of Soundness

The soundness of the compilation process follows from the modular verification and syntactic well-formedness of the input program. We present a sketch of the proof, with key theorems describing the verification of individual components, the soundness of compound module linking, and the logical properties of substitution.

It makes no sense to consider the provability of a syntactically ill-formed program. For instance, introducing two different definitions for the same function name results in unsoundness. All of the provability judgments in figures 10 (for ACL2) and 7 (for Modular ACL2) are implicitly predicated on their constituents being well-formed. ACL2 requires that all stub, function, and theorem names be distinct and have no forward references. Modular ACL2 requires the same of interface and module names. Atomic modules must import or export each external name by at most one interface (not counting extensions). Compound modules must obey the same rule, which in turn requires nominal linking, i.e. definitions resolved by linking must be imported and exported via the same interface. Soundness requires all of these properties to hold true at all stages of verification and linking; however, for space reasons we do not present a formal treatment of them here.

For our proof, we introduce the abbreviation $\Gamma_i \vdash_m \Gamma_m$, which means $\vdash_s \overrightarrow{\textsf{obligations}(\Gamma_i, mod)}$ when $\Gamma_m = \overrightarrow{mod}$.

Main Theorem. *If* $\vdash_p mprog$*, then* $\vdash_s \textsf{execute}(mprog)$.

Here we state the soundness of Modular ACL2 with respect to ACL2: if the theorem prover verifies the proof obligations of each module in a syntactically well-formed program, then the linked executable code is provable as well.

Proof. The main theorem generalizes to lemma 1, which reasons component-wise about the body of $mprog$. □

Lemma 1. *If* $\Gamma_i \vdash_m \Gamma_m$ *and* $\Gamma_i \vdash_c \overrightarrow{comp}$*, then* $\Gamma_e \vdash_t \textsf{compile}(\Gamma_i, \Gamma_m, \Gamma_r, \overrightarrow{comp})$.

Note that Γ_r affects only top-level expressions, and thus plays no role in logical soundness, just well-formedness.

Proof. By induction over $\overrightarrow{comp}$, and by cases on its components.

Case ϵ**:** Trivial.

Case $ifc\ \overrightarrow{comp}$**:** In this case, ifc is simply moved from the sequence of components to the interface environment (in both verification and compilation). We show that adding ifc to Γ_i entails $\Gamma_i\ ifc \vdash_m \Gamma_m$: the proof obligations of each module are unchanged, as they make no reference to ifc. By PFIFC, we reduce the proof to the inductive hypothesis.

Case $mod\ \overrightarrow{comp}$**:** In this case, mod is a verified atomic module. We record its verification by storing it in the module environment. Adding mod to Γ_m trivially entails $\Gamma_i \vdash_m \Gamma_m\ mod$. Using PFMOD, the proof reduces to the inductive hypothesis.

Case (link n $(n_1\ n_2)$**)** $\overrightarrow{comp}$**:** Here we must prove that linking two verified constituents produces a verified module. In lemma 2 below, we show that $\textsf{link}(\Gamma_i, n, \Gamma_m(n_1), \Gamma_m(n_2))$ is provable in Γ_e and Γ_i. We finish with PFLINK and the inductive hypothesis.

Case $inv\ \overrightarrow{comp}$**:** Invocation renames module bodies. We show in lemma 4 that this is logically equivalent to renaming the resulting definitions. In lemma 6, we also show that the definitions' provability is preserved. By rule PFINV the proof reduces to the inductive hypothesis.

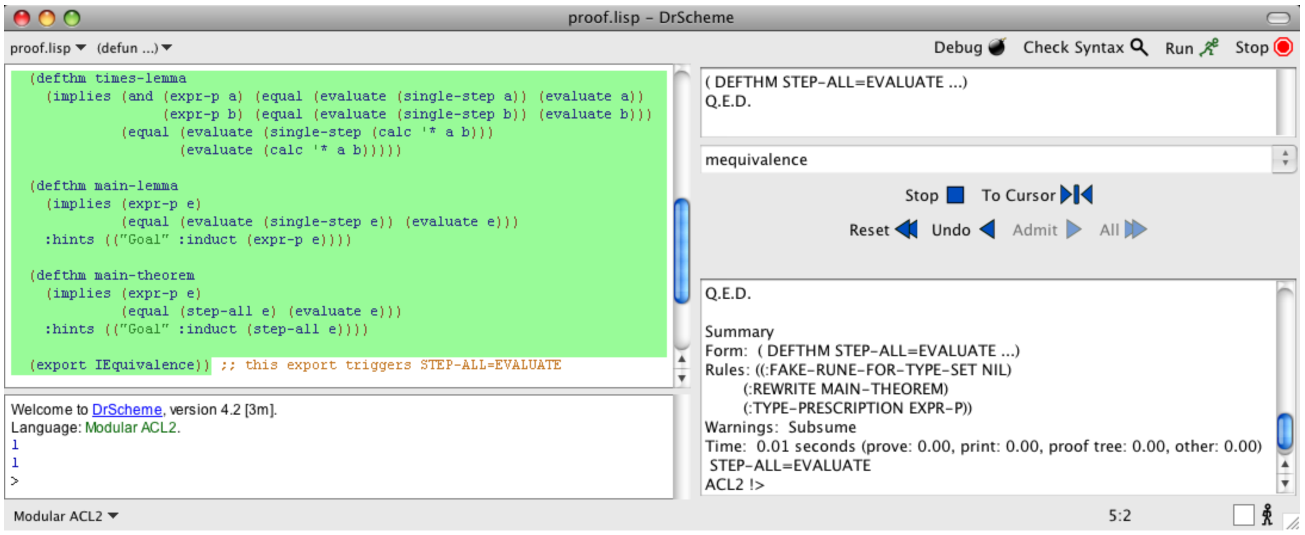
Case $expr\ \overrightarrow{comp}$**:** By PFCEXPR and the inductive hypothesis. □

Lemma 2. $\vdash_s \textsf{obligations}(\Gamma_i, \textsf{link}(\Gamma_i, n, mod_1, mod_2))$ *holds if* $\vdash_s \textsf{obligations}(\Gamma_i, mod_1)$ *and* $\vdash_s \textsf{obligations}(\Gamma_i, mod_2)$.

The crux of our proof is to establish the soundness of linking.

Proof. Let mod_1 be (**module** $n_1\ \overrightarrow{body_1}$) and mod_2 be (**module** $n_2\ \overrightarrow{body_2}$). By definition, $\textsf{link}(\Gamma_i, n, mod_1, mod_2)$ is (**module** $n\ \overrightarrow{body_3}\ \overrightarrow{body_4}$) where $\overrightarrow{body_3} = \textsf{rename}(\overrightarrow{body_1})$ and $\overrightarrow{body_4} = \textsf{resolve}(\overrightarrow{body_3}, \textsf{rename}(\overrightarrow{body_2}))$. By rule PFPROG and the definition of *obligations*, we must prove:

$$\Gamma_e^0 \vdash_t \overrightarrow{\textsf{verify}(\Gamma_i, body_3)}\ \overrightarrow{\textsf{verify}(\Gamma_i, body_4)}$$



```
(defthm times-lemma
  (implies (and (expr-p a) (equal (evaluate (single-step a)) (evaluate a))
                (expr-p b) (equal (evaluate (single-step b)) (evaluate b)))
           (equal (evaluate (single-step (calc '* a b)))
                  (evaluate (calc '* a b)))))

(defthm main-lemma
  (implies (expr-p e)
           (equal (evaluate (single-step e)) (evaluate e)))
  :hints (("Goal" :induct (expr-p e))))

(defthm main-theorem
  (implies (expr-p e)
           (equal (step-all e) (evaluate e)))
  :hints (("Goal" :induct (step-all e))))

(export IEquivalence)) ;; this export triggers STEP-ALL=EVALUATE
```

```
Welcome to DrScheme, version 4.2 [3m].
Language: Modular ACL2.
1
1
>
```

```
( DEFTHM STEP-ALL=EVALUATE ...)
Q.E.D.
```

```
Q.E.D.

Summary
Form:  ( DEFTHM STEP-ALL=EVALUATE ...)
Rules: ((:FAKE-RUNE-FOR-TYPE-SET NIL)
        (:REWRITE MAIN-THEOREM)
        (:TYPE-PRESCRIPTION EXPR-P))
Warnings:  Subsume
Time:  0.01 seconds (prove: 0.00, print: 0.00, proof tree: 0.00, other: 0.00)
 STEP-ALL=EVALUATE
ACL2 !>
```

Figure 13. Verification of the interpreter experiment in Dracula.

We apply lemma 6 to show that $\overrightarrow{body_1}$'s provability entails $\overrightarrow{body_3}$'s. Then by lemma 8, we can focus on the second set of obligations and it suffices to prove:

$$\Gamma_e^0\ \overline{\mathit{theory}(\mathit{verify}(\Gamma_i, \overrightarrow{body_3}))} \vdash_t \overline{\mathit{verify}(\Gamma_i, \overrightarrow{body_4})}$$

Lemma 6, applied to $\overrightarrow{body_2}$, shows that $\mathit{rename}(\overrightarrow{body_2})$ is provable. Now we must demonstrate that *resolve* correctly discharges the assumptions of the second module. We call this lemma 3. □

Lemma 3. *If* $\Gamma_e \vdash_t \overline{\mathit{verify}(\Gamma_i, \overrightarrow{body_2})}$ *and* $\overrightarrow{body_3} = \mathit{resolve}(\overrightarrow{body_1}, \overrightarrow{body_2})$*, then* $\Gamma_e\ \overline{\mathit{theory}(\mathit{verify}(\Gamma_i, \overrightarrow{body_1}))} \vdash_t \mathit{verify}(\Gamma_i, \overrightarrow{body_3})$.

Proof. By induction on the length of $\overrightarrow{body_2}$ and by cases on its first element. Only the case for imports is nontrivial; we omit the rest. Assume $\overrightarrow{body_2} = $ (**import** n $\overrightarrow{(f\ f_1)}$) $\overrightarrow{body}$. Let $\Gamma_i(n) = $ (**interface** n $\overrightarrow{spec}$). We proceed by cases on $\overrightarrow{body_1}$.

Case (export n $\overrightarrow{(f\ f_2)}$**)** $\in \overrightarrow{body_1}$**:** In this case, *resolve* removes the import from $\overrightarrow{body_2}$ and renames the remainder based on the export from $\overrightarrow{body_1}$. We show that this transformation is sound: after renaming, the logical obligations of the export fill in precisely the logical assumptions of the removed import. We rely on lemma 9 to equate assumptions with obligations. The proof proceeds by equational reasoning in figure 12.

Case (import n $\overrightarrow{(f\ f_2)}$**)** $\in \overrightarrow{body_1}$**:** This case proceeds as above, but without the appeal to lemma 9.

Case $n \not\in \overrightarrow{body_1}$**:** Trivial. □

Lemma 4. *If* $\Gamma_e \vdash_t \mathit{obligations}(\Gamma_i, \epsilon, \mathit{rename}(\overrightarrow{body}))$*, then* $\Gamma_e \vdash_t \mathit{rename}(\mathit{obligations}(\Gamma_i, \epsilon, \overrightarrow{body}))$.

Lemma 5. $\mathit{verify}(\Gamma_i, \overrightarrow{body})[\overrightarrow{f_1 = f_2}] = \mathit{verify}(\Gamma_i, \overrightarrow{body}[\overrightarrow{f_1 = f_2}])$.

Proof. By the definitions of *verify* and substitution. Lemma 4 follows as a corollary. □

Lemma 6. *If* $\Gamma_e \vdash_t \overrightarrow{defn}$*, then* $\Gamma_e \vdash_t \mathit{rename}(\overrightarrow{defn})$.

Lemma 7. *If* $\Gamma_e \vdash_t \overrightarrow{term}$ *and* $f \not\in \overrightarrow{term}$*, then* $\Gamma_e \vdash_t \overrightarrow{term}[\overrightarrow{f_0 = f}]$.

Proof. This follows from the proof of soundness of simple functional instantiations [15]. Lemma 6 follows as a corollary. □

Lemma 8. $\Gamma_e \vdash_t \overrightarrow{defn_1}\ \overrightarrow{defn_2}$ *if and only if both* $\Gamma_e \vdash_t \overrightarrow{defn_1}$ *and* $\Gamma_e\ \overline{\mathit{theory}(\overrightarrow{defn_1})} \vdash_t \overrightarrow{defn_2}$

Proof. Both directions follow by induction over $\overrightarrow{defn_1}$. □

Lemma 9. $\mathit{theory}(\mathit{assume}(\mathit{spec})) = \mathit{theory}(\mathit{assert}(\mathit{spec}))$.

Proof. By definitions of *theory*, *assume*, and *assert*. □

6.3 Expressivity Proof

Expressivity means that any decomposition of a proof in the core grammar of ACL2 into contiguous blocks of definitions can be represented as a set of modules in Modular ACL2, and the theorem prover can verify the modular program if it can verify the original.

The translation from a proof to modules is straightforward. Each section of the proof corresponds to one atomic module and one interface. The interface contains a manifest function for every function and a contract for every conjecture. Essentially, the interface exactly reconstructs the sequence of definitions. Each interface extends those before it. The corresponding atomic module imports the previous interfaces and exports the matching interface. The module need not contain any definitions, as all components of the specifications are concrete (assuming no stubs in the source proof). The modular program concludes by progressively linking the modules together in order; the final compound module consisting of all the proof sections can be invoked to run the original ACL2 program.

This translation relies on the one-to-one mapping between core ACL2 definitions and Modular ACL2 specifications (other than **extend**). Each interface expresses precisely the definitions of one section of the proof. The proof obligation of each atomic modules starts with an import that recreates the logical environment of the proof section as it was in the ACL2 version, and it concludes with an export that entails the same logical verification as well.

The formal proof of correctness of this translation follows by induction over the sequence of modules. The proof obligations of the modules can be shown to be a partitioning of the proof obligations of the original ACL2 program by the argument above.

	Optimizations				
Lemmas	**Mono**	**20**	**40**	**80**	**Mod**
modulo/range	0.04	0.04	0.04	0.04	0.04
random-nat/range	0.05	0.05	0.05	0.05	0.05
worm-turn/uncrossed	0.19	0.19	0.19	0.19	0.04
game-tick/gamep	142.88	19.08	6.35	2.00	0.01
game-key/gamep	22.73	2.92	2.93	1.85	0.01
game-tick/in-bounds	136.67	19.51	6.13	2.28	0.01
game-key/in-bounds	22.63	2.87	2.88	1.22	0.01
game-tick/uncrossed	320.84	16.19	6.23	2.29	0.02
game-key/uncrossed	51.86	5.93	5.96	2.14	0.02
connected-gamep/gamep	60.94	0.00	0.00	0.00	0.01
Improvements	–	9	21	52	–

Figure 14. Benchmarks of Worm game proof.

Lemmas	**Mono**	**Mod**
single-step-expr	†0.01	0.01
single-step-calc	†0.03	0.01
single-step/evaluate/+	0.96	0.06
single-step/evaluate/-	0.92	0.06
single-step/evaluate/*	0.94	0.06
single-step/evaluate	3.08	0.02
step-all=evaluate	†0.00	0.00

†failed without additional hints

Figure 15. Benchmarks of interpreter proof.

DFS Lemmas	**NLG**	**ELG**	**Mod**
subsetp-equal-prefix	0.01	0.01	0.02
neighbors/nodes	‡0.00	0.00	0.00
choose-route/pathp	0.20	0.18	0.17
find-route/pathp	0.02	0.01	0.04
BFS Lemmas	**NLG**	**ELG**	**Mod**
choose-route/pathp	0.01	0.01	0.01
search-routes/pathp	0.02	0.02	0.02
find-route/pathp	0.00	0.00	0.01

‡failed without additional hints

Figure 16. Benchmarks of graph search proofs.

	Mono	**Mod**
Worm	134.77	135.40
Interpreters	115.67	116.37
DFS/NLG	9.03	9.00
DFS/ELG	13.82	13.88
BFS/NLG	158.19	158.11
BFS/ELG	444.28	445.15

Figure 17. Benchmarks of monolithic vs. modular execution.

7. Implementation and Benchmarks

In our previous report [5], we describe a prototype implementation of Modular ACL2 and present experiments comparing modular proofs to the same proof in monolithic form. Since that writing, the implementation of Modular ACL2 has been updated with translucent interfaces and released for public download as part of Dracula.

Dracula [26], the dialect of ACL2 for the DrScheme programming environment [7], includes a simulation of Modular ACL2's runtime behavior and an interface to the theorem prover for logical verification. Dracula links and runs modules in its own simulation based on figure 9. Each module is compiled [3, 8] to a closure that extends a table of imports with a set of exports; compound modules are constructed by function composition. Module invocation calls a closure with an empty mapping and exposes the resulting exports.

The implementation also generates proof obligations for each module based on figures 7 and 8. The compiled syntax of each module is annotated with the proof obligation; Dracula extracts the annotations and sends them to ACL2 as directed by the user. Dracula automatically resets the theorem prover upon starting each module.

Figure 13 shows the final fragment of the interpreter experiment from section 5 in Dracula. On the left, the definitions window (top) contains the program and the interactions window (bottom) shows the result of the top level calls to evaluate and step-all. On the right, there are a proof summary (top), proof controls (middle), and a transcript of the theorem prover's output (bottom). The proof summary, transcript, and definitions window highlighting all indicate that the IEquivalence module is successfully verified.

7.1 Proof Experiments

In support of our claim concerning modular theorem proving, we present benchmarks for our three experiments, comparing their modular proofs to monolothic versions. The latter were produced by concatenating all the proof obligations together in one file. For each program, we compare both verification and execution times.[2]

As shown in our prior work, the video game experiment demonstrates the search space explosion common to monolithic proof attempts. As the theorem prover accrues logical rules, the time to prove lemmas increases. The modular version, however, starts with a "clean slate" after each module boundary. By the end of the proof, the monolithic takes several orders of magnitude longer for key lemmas than the worst times of the modular version. Figure 14 shows this discrepancy in the "Mono" and "Mod" columns, listing the slowest lemmas for both proof attempts.

Professional ACL2 programmers are able to tune the performance of monolithic ACL2 proofs by giving "hints" that disable appropriate logical rules. To compare the benefits of modularity to these hints, we optimized the monolithic proof of the "Worm" game by the following process. At each step, we identified the slowest lemma and used ACL2's **accumulated-persistence** profiling tool to identify which rules the theorem prover was spending the most time on during the proof attempt. We added a hint to the lemma instructing the theorem prover to disable the most time-consuming rule during future proof attempts, then tried it again and measured the difference. We kept the hints that improved the verification speed and discarded those that made it take longer (or fail). The middle columns of figure 14 show the results of the first 20, 40, and 80 attempts to disable rules in the 83-lemma proof. Nine successes out of the first 20 attempts dramatically improve the prover's running time, improving some of the slowest lemmas by a factor of ten. The next 60 attempts have less of an impact; most of the slow lemmas remain at a few seconds of theorem proving time, compared to the modular proof in which all lemmas run in 0.05 seconds or less.

Benchmarks of the graph and interpreter experiments show similar results; see figure 15 and 16. Though the proof times do not show as great a difference, monolithic proofs in both experiments fail to complete when converted to monolithic form. In the interpreter experiment, three lemmas in the components for single-step

[2] All times are measured in seconds based on the average of five trials. All benchmarks were performed on a 2 GHz dual-core MacBook with 4GB RAM running Mac OS X 10.5.6. We used ACL2 3.5 built on Clozure Common Lisp 1.3 and Dracula 8.2 in DrScheme 4.2.

and equivalence require hints disabling calc-p and related functions to escape apparent infinite loops during verification. For the graph experiment, we constructed four monolithic proofs: one for each combination of depth-first search (DFS) or breadth-first search (BFS) with edge list graphs (ELG) or neighbor list graphs (ELG). We timed the graph search portions of these against the modular proofs of breadth-first and depth-first search. The monolithic proof of depth-first search using neighbor lists failed until an extra hint was given to the neighbor/nodes lemma. These experiments demonstrates the robustness of modular proofs, as they do not fail when combined with other components, nor do they need to be duplicated for multiple instantiations.

Finally, in order to demonstrate that modularization does not add any appreciable overhead to the running time of ACL2 programs, we benchmarked all three experiments on problems taking from a few seconds to a few minutes: an exploration of all "Worm" game states reachable in 20 steps, evaluation of an expression tree 13 nodes deep, and traversal of a graph containing a 7-clique. Figure 17 shows the results of the trials; the execution times were comparable in all cases.

8. Conclusions

Our prior work extended ACL2 with a module system that provides abstraction, reusability, and lexical scope beyond that available in ACL2, but it had a serious shortcoming. Specifically, the language could not communicate induction schemes between components.

Modular ACL2 now incorporates manifest functions, which express induction schemes as recursive functions in the manner ACL2 is tailored to expect. Furthermore, the module system is accompanied by a formal proof of soundness, guaranteeing that the verified properties of each component individually remain true when components are linked and executed collectively, and an expressivity proof showing that all core ACL2 constructs can be communicated across abstraction boundaries.

Our experiments show that the module system expresses programs and properties of moderate complexity, that it reduces the programmer's burden of theorem proving and proof optimization, and that it increases the expressivity of the proof language to include new degrees of abstraction.

Acknowledgments

Thanks to Rex Page for adopting Modular ACL2 in the classroom; to Pete Manolios, Peter Dillinger, Matt Kaufmann, J Moore, Ruben Gamboa, and the ACL2 community for many helpful insights and suggestions, to Sky O'Mara for help with the implementation of Modular ACL2; and to T. Stephen Strickland, Christos Dimoulas, and the anonymous referees for comments on early drafts.

References

[1] Chrzaszcz, J. Implementing modules in the Coq system. In *Proc. 16th International Conference on Theorem Proving in Higher Order Logics*, p. 270–286. Springer, 2003.

[2] Courant, J. $\mathcal{MC}_2$: a module calculus for pure type systems. *J. Functional Programming*, 17(3):287–352, 2007.

[3] Culpepper, R., S. Tobin-Hochstadt and M. Flatt. Advanced macrology and the implementation of Typed Scheme. In *Proc. 8th Workshop on Scheme and Functional Programming*, p. 1–14. ACM Press, 2007.

[4] Dreyer, D. and A. Rossberg. Mixin' up the ML module system. In *Proc. 13th ACM SIGPLAN International Conference on Functional Programming*, p. 307–320. ACM Press, 2008.

[5] Eastlund, C. and M. Felleisen. Toward a practical module system for ACL2. In *Proc. 11th International Symposium on Practical Aspects of Declarative Languages*, p. 46–60. Springer, 2009.

[6] Farmer, W. M., J. D. Guttman and F. J. Thayer. IMPS: An interactive mathematical proof system. *J. Automated Reasoning*, 11(2):213–248, 1993.

[7] Findler, R. B., J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler and M. Felleisen. DrScheme: A programming environment for Scheme. *J. Functional Programming*, 12(2):159–182, 2002.

[8] Flatt, M. Composable and compilable macros: You want it *when?* In *Proc. 7th ACM SIGPLAN International Conference on Functional Programming*, p. 72–83. ACM Press, 2002.

[9] Flatt, M. and M. Felleisen. Units: cool modules for HOT languages. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, p. 236–248. ACM Press, 1998.

[10] Harper, R. and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proc. 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, p. 123–137. ACM Press, 1994.

[11] Kammüller, F., M. Wenzel and L. C. Paulson. Locales: a sectioning concept for Isabelle. In *Proc. 12th International Conference on Theorem Proving in Higher Order Logics*, p. 149–166. Springer, 1999.

[12] Kaufmann, M., P. Manolios and J. S. Moore. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, 2000.

[13] Kaufmann, M., P. Manolios and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[14] Kaufmann, M. and J. S. Moore. A precise description of the ACL2 logic. Technical report, University of Texas at Austin, 1998.

[15] Kaufmann, M. and J. S. Moore. Structured theory development for a mechanized logic. *J. Automated Reasoning*, 26(2):161–203, 2001.

[16] Leroy, X. Manifest types, modules, and separate compilation. In *Proc. 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, p. 109–122. ACM Press, 1994.

[17] Martín-Mateos, F. J., J. A. Alonso, M. J. Hidalgo and J. L. Ruiz-Reina. A generic instantiation tool and a case study: a generic multiset theory. In *Proc. 3rd International Workshop on the ACL2 Theorem Prover and its Applications*, p. 188–201. ACM Press, 2002.

[18] Milner, R., M. Tofte and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[19] Owens, S. and M. Flatt. From structures and functors to modules and units. In *Proc. 11th ACM SIGPLAN International Conference on Functional Programming*, p. 87–98. ACM Press, 2006.

[20] Padget, J. et. al. Desiderata for the standardisation of LISP. In *Proc. 1986 ACM Conference on LISP and Functional Programming*, p. 54–66. ACM Press, 1986.

[21] Page, R. Engineering software correctness. *J. Functional Programming*, 17(6):675–686, 2007.

[22] Page, R., C. Eastlund and M. Felleisen. Functional programming and theorem proving for undergraduates: a progress report. In *Proc. 2008 Workshop on Functional and Declarative Programming in Education*, p. 21–30. ACM Press, 2008.

[23] Sannella, D. Formal program development in Extended ML for the working programmer. In *Proc. 3rd BCS/FACS Workshop on Refinement*, p. 99–130. Springer, 1991.

[24] Steele Jr., G. L. *Common Lisp—The Language*. Digital Press, 1984.

[25] The Coq Development Team. *The Coq Proof Assistant Reference Manual*, 2009. `http://coq.inria.fr/coq/distrib/current/refman/`.

[26] Vaillancourt, D., R. Page and M. Felleisen. ACL2 in DrScheme. In *Proc. 6th International Workshop on the ACL2 Theorem Prover and its Applications*, p. 107–116. ACM Press, 2006.

[27] Wirth, N. *Programming in Modula-2*. Springer, 1983.