Modular Proof Development in ACL2

A dissertation presented

by

Carl Eastlund

to the Faculty of the Graduate School

of the College of Computer and Information Science

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Northeastern University

Boston, Massachusetts

December, 2012

Thanks to everyone.

Thanks to my parents for getting me interested in reading, learning, and programming from as long ago as I can remember. Thanks for a million other things I can't even begin to list.

Thanks to my brother Paul for always reminding a quieter older brother what energy, ambition, and competitiveness are for. Even if I didn't always appreciate it at the time. Perhaps especially because I didn't.

Thanks to my nieces Ruthie and Claire and my nephew Paul, just for being cute and mispronouncing words. Thanks to their expected new sister, I look forward to meeting you.

Thanks to all my friends in high school and college for constantly keeping me challenged.

Thanks especially to Tim and Simon for keeping in touch all these years and for beating me to the Ph.D., thus reminding me to get finished, already.

Thanks to Triften for all the Sunday-morning video game sessions. I might have been able to graduate without them, but it would hardly have been worth it.

Thanks to my fellow students in graduate school for all the cooperation, collaboration, and commiseration. Thanks especially to Sam Tobin-Hochstadt and Stevie Strickland, my fellow racketeers and partners in crime.

Thanks to Zoe Zhang, Ken McGrady, Sky O'Mara, and Carter Schonwald for all their help developing and testing Dracula, the platform for all my research. Thanks to Dale Vaillancourt for getting all this started.

Thanks to Matt Kaufmann, J Moore, and the rest of the ACL2 community for their support and acceptance of this quirky outsider with odd new ideas about how their mature, successful, and award-winning software really ought to work.

Thanks to Dan Friedman for seeing new potential in my work, for seeing simplicity and elegance in everything, and for patiently putting boundless energy and exuberance on hold for two and a half years so I could finish this document.

Thanks to Rex Page, Ruben Gamboa, Pete Manolios, and Riccardo Pucella for their valuable feedback on this work, without which it could not have been as good as it is.

And thanks finally to Matthias Felleisen. Nine and a half years is an incredible commitment to putting up with me and supporting me. I would not have made it without you.

Contents

List of Figures

Chapter 1

Introduction

ACL2 is the successor to Nqthm, the original Boyer-Moore theorem prover. In 1989, Boyer, Moore, and Kaufmann set out to design a more practical theorem prover suited to verifying real-world hardware and software artifacts. The resulting software has gained significant success and popularity among industrial clients.

The ACL2 theorem prover comprises both a programming language and a logic, aptly described by its acronym: "A Computational Logic for Applicative Common Lisp". "Applicative Common Lisp" refers to the first-order, side-effect free variant of Common Lisp that serves as ACL2's programming language. The "Computational Logic" describes an automated theorem prover for first-order logic over the domain of recursive functions in the aforementioned language.

In addition to ACL2's core constructs—recursive functions and theorems about them—there are several meta-proving tools that extend the power of the language and the logic, allowing developers to manage complicated verification tasks. From its Common Lisp heritage, ACL2 inherits macros for syntactic abstraction and packages for namespace management. Logically, ACL2 has tools to divide programs by file, to nest encapsulated abstractions and later reinstantiate them with concrete definitions, to defer verification by "skipping" certain proof obligations, and to extend the logic with new axioms.

For all these capabilities, there are several program development techniques ACL2 does not natively support. The logic indirectly supports second-order universal quantification via encapsulation (second-order existential quantification) and functional instantiation (reusing theorems that reason about an existential), but does not support direct universal quantification without an existential witness. Top-down development requires editing a partially-verified program to remove "skip-proofs" annotations; the final steps in verification do not constitute a verified artifact on their own. Packages and books do not protect their contents from namespace collisions; encapsulated abstractions do, but at the cost of executability. Finally, ACL2's macros are unhygienic, meaning they do not respect lexical scope.

Fortunately, ACL2's existing tools show many dimensions along which it can be extended to accommodate these and other features. ACL2 admits to syntactic extension via macros; to representing new features dually, split between concrete and abstract representations, as with encapsulation blocks; and to second-order reasoning as with functional instantiation. Based on these avenues for extending the language, I arrive at my thesis.

> The language of ACL2 can be extended to express robust modular and syntactic abstractions without changing ACL2's logic or theorem prover.

To demonstrate this thesis, I present Refined ACL2. Refined ACL2 is a proof language combining the programmatic and logical constructs of ACL2 with ML's modularity mechanisms and Racket's syntactic abstractions. To illustrate the design process leading up to Refined ACL2, I also describe Modular ACL2 and Hygienic ACL2, two predecessor languages of Refined ACL2 that separately introduce modules and hygienic macros to ACL2. Along with these languages, I present experiments validating Refined ACL2's utility for developing ACL2 proofs and a soundness theorem for the Refined ACL2 module system.

1.1 History

This research started with Dracula. Prof. Page (University of Oklahoma) requested ACL2 in DrRacket (then DrScheme) as a better environment for teaching formal verification. Dale Vaillancourt designed and implemented the original Dracula, an interface for running a simulation of ACL2 execution in DrScheme and relaying proof obligations to ACL2. Page also requested a way to demonstrate modular programming development, which ACL2 only supports through a fragile combination of many features. The goal was to provide a system in which components of an ACL2 model can be specified, compiled, and verified separately, but ultimately executed as a whole without duplicating effort (of the developer, compiler, or theorem proving engine). I took on the task of implementing [Eastlund and Felleisen 2009b] this system.

The first iteration of the module system is based on the linking system of the Racket unit system, with bindings determined as in mixin modules. It provides a way to specify functions via signatures and contracts, a way to implement specifications in terms of other specifications, and ways to link implementations together and ultimately execute them. Our first model, described in more detail in appendix A, implemented the bare minimum necessary for these features. Subsequent versions of the language could intermingle interface and module definitions, could link across multiple interfaces at once, accumulated exported

bindings of executable modules at the top level, and allowed interfaces to depend on one another.

A series of short case studies demonstrated the resulting system adequate for developing proof abstractions and restricting the theorem prover's search efforts. However, the specification language of signatures and contracts was not expressive enough to describe new induction schema. For instance, one case study introduced a predicate recognizing arithmetic expressions with integers and binary operators. In ACL2, the theorem prover deduces structural induction over arithmetic expressions from the recursion scheme used to define the predicate. Our language of contracts describes the predicate's results but not its recursion scheme. Clients of the expression datatype, therefore, were unable to perform structural induction.

Our second iteration of the module system introduced transparent functions, describing induction in terms of recursive function definitions as done natively by ACL2. The same case studies were performed in the second system, this time with greater success with novel induction schemes. The system also came with an expressivity proof, showing that the module system could faithfully reproduce any model (in its contained subset of ACL2) split into modules at arbitrary points.

Once modules performed adequately in our small case studies, we turned to the one abstraction feature of ACL2 that we had never integrated with Dracula: macros. On one hand, macros are a useful tool for enriching the core language of ACL2, understood by the theorem prover, with useful tools for programmers that the theorem prover need not be extended to reason about. On the other hand, ACL2's macros are unhygienic; they do not preserve default assumptions about lexical scope in the surface program. This makes them difficult to trust, and a significant step down from Racket's advanced hygienic macro system.

Our first macro system, Hygienic ACL2, attempts to bring hygienic macro expansion to ACL2 while preserving compatibility with its existing libraries and macros. Rather than devise a new macro system, we augment the values of ACL2 with expansion metadata, invisible to the logic but usable by the compiler, and use the metadata to give a hygienic semantics to existing macro definitions. We base our definition of hygiene on the scope of ACL2 and on compatibility with existing macro practices.

In 2009, Ms. Yue "Zoe" Zhang (M.S. 2010, Northeastern University) ran a large-scale evaluation of Modular ACL2. She chose to model the Racket bytecode verifier and attempt to prove its soundness. Although she did not complete the verification effort, her project suggested several opportunities for improvement in Modular ACL2.

The primary difficulties, aside from the complexity of the bytecode verifier itself, were

structure specifications, list type abstraction, (mutually) recursive datatypes, and mutual induction. Modular ACL2 cannot abstract over structure or recursive datatype definitions, as they have a non-uniform format. Constructing the theories necessary to represent structures abstractly and recursive datatypes in a form that permits induction and recursion took a disproportionate amount of time and space. Furthermore, any changes to the basic template for either induced a manual change to each implementation. Mutual induction did not balloon out of proportion, but provided similar syntactic difficulties.

While Modular ACL2 can abstract over list types, instantiations do not expose their element types, which basically renders the abstraction useless. The module system's scope and linking mechanisms were also never augmented to allow multiple instances of one specification, so list types could not be combined.

Essentially, Modular ACL2 is useful for segmenting the large components of a model, but does not come with good facilities for building the low level tools of representation and reasoning in ACL2.

Our response to these difficulties is to design a language combining the modular proof abstractions of Modular ACL2, the flexibility of linking, nesting, and specializing of the ML module system, and the syntactic abstractions of the Racket macro system. We anticipate that nesting and specializing modules will make list abstractions practical, while hygienic macros will allow automation for mutual induction, recursive datatypes, and structure datatypes.

1.2 Roadmap

The following chapter explains Modular ACL2, our initial design for an ACL2 module system. The discussion begins in section 2.1 with the difficulties inherent to modular proof development in ACL2. Section 2.2 provides an overview of Modular ACL2's features by example. Both semantic translations from Modular ACL2 to ACL2 are described in subsequent sections—section 2.3 for the executable semantics and section 2.4 for the logical semantics. The chapter concludes with a proof of soundness relating the logical and executable semantics.

Our adaptation of hygienic macro expansion to existing ACL2 programs is presented in chapter 3. We provide several examples illustrating how ACL2's unhygienic macros can be problematic in section 3.1. Section 3.2 describes our interpretation of hygiene for the particular syntactic forms of ACL2, including separate function and value namespaces, implicit quantification, and local scope in encapsulated proofs. Extending this description, we present the details of Hygienic ACL2's expansion algorithm in section 3.3. We evaluate

the impact of hygienic expansion on the existing body of ACL2 macros in section 3.4.

Chapter 4 discusses a case study in which we attempt to use Modular ACL2 to model and verify the Racket virtual machine and bytecode verifier, in order to report on Modular ACL2's utility in practice. Section 4.1 describes a simple subset of the Racket bytecode language and verification algorithm. In section 4.2 we present our approach to proving the verifier's soundness, followed by an analysis of which Modular ACL2 features worked and which didn't in section 4.3.

In response to the case study above, we present Refined ACL2, a language combining ML modules, Racket macros, and the logic of ACL2 to overcome the drawbacks we uncovered. Chapter 5 begins with a description of Refined ACL2 and an example program. Macro expansion in our new language is described in section 5.2. Sections 5.3, 5.4, and 5.5 present the static, logical, and executable semantics of Refined ACL2, respectively. We prove the soundness of our logical semantics with respect to the executable semantics in section 5.6. The chapter ends with a discussion of the implementation details for Refined ACL2 in both Racket and ACL2.

The support of our thesis draws to a conclusion in chapter 6, in which we discuss our accomplishments and consider future directions.

Chapter 2

Modules

Modular ACL2[1] augments ACL2 with modules that can be verified independently, then linked and run as a whole. We present the motivation and design of Modular ACL2, its semantics, and a proof demonstrating that the linking process preserves the theorems proved about individual modules.

2.1 Motivation

For our purposes, 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 2.1 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.

[1] This work has been published as Eastlund and Felleisen [2009a,b].

```
(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 2.1: A finite set representation in ACL2.

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 Kaufmann and Moore [2001]. 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 [Padget, J. et al. 1986]. 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. [2002] 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 [Kaufmann et al. 2000] 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.

2.2 Design

Large proofs in ACL2 require a consolidated system for specification, abstraction, and the management of namespaces and components. To that end 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 in figure 2.1 can be rewritten as a Modular ACL2 program in which add and add-all are specified and implemented separately; see figure 2.2. 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.

```
(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 2.2: A finite set representation in Modular ACL2.

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 2.1.

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 automatically 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 [Kaufmann et al. 2000], 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 highlights one of ACL2's key reasoning mechanisms: induction schemes inferred from function definitions. The experiment specifies the equivalence of two interpreters via four interfaces, shown partially in figure 2.3.

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

This section presents the formal definition 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.

2.2.1 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

```
(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 2.3: Excerpts from interfaces in the interpreter experiment.

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 [1994].

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 2.5. It extends the core grammar of ACL2 in figure 2.4. 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 $\overrightarrow{X}^n$.

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

Figure 2.4: 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.[2] ACL2 includes two variable namespaces: one for functions and conjectures (f) and another for function parameters and local variables (x). Modular ACL2 adds a third namespace for interfaces and modules (n).

Modular ACL2 programs (*mprog*) consist of a sequence of components. A component may be an interface (*ifc*), atomic module (*mod*), compound module (*link*), module invo-

[2]The **skip-proofs** form may admit unsound conjectures, and is usually reserved for intermediate stages of proof development. See section 2.4.1.

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

Figure 2.5: The grammar of Modular ACL2.

cation (*inv*), or top level expression (*expr*). 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 links together two named constituent modules.

An interface may specify manifest functions (*fun*), opaque signatures (*sig*), contracts (*con*), or dependencies (*ext*). 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 (*defn*), imports (*im*), and exports (*ex*). 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 chapter, 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

```
(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 2.6: Excerpts from modified interpreter interfaces.

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 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 2.3) with manifest functions. Figure 2.6 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 can use 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 immediately using the manifest definitions of step-all and expr-p to reason inductively about step-all and evaluate, respectively.

In general, manifest functions are useful whenever the contents of one module introduce a pattern of recursion that another module needs to reason about. The pattern of recursion

may be a data structure (as with expr-p) or an algorithm (as with step-all). 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.

2.3 Execution

The executable semantics for Modular ACL2 turns a whole Modular ACL2 program into a single ACL2 program intended for execution in ACL2's program mode. Our soundness theorem guarantees that if the verification obligation for the whole program can be proved, the theorems in the executable form are logically sound as well. Explicitly verifying them would therefore be redundant.

2.3.1 Executing Modules

The execution of Modular ACL2 programs is defined by the metafunction *execute*, shown in figure 2.7, 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 (Γ_i), modules (Γ_m), and renamings (Γ_r), 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.

$$
\begin{array}{l}
\mathit{execute} : \mathit{mprog} \to \mathit{prog} \\
\quad \mathit{execute}(\mathit{mprog}) \qquad\qquad = \mathit{compile}(\epsilon, \epsilon, \epsilon, \mathit{mprog}) \\
\\
\mathit{compile} : \Gamma_i, \Gamma_m, \Gamma_r, \overrightarrow{\mathit{comp}} \to \overrightarrow{\mathit{term}} \\
\quad \mathit{compile}(\Gamma_i, \Gamma_m, \Gamma_r, \mathit{ifc}\ \overrightarrow{\mathit{comp}}) \qquad = \mathit{compile}(\Gamma_i\ \mathit{ifc}, \Gamma_m, \Gamma_r, \overrightarrow{\mathit{comp}}) \\
\quad \mathit{compile}(\Gamma_i, \Gamma_m, \Gamma_r, \mathit{mod}\ \overrightarrow{\mathit{comp}}) \qquad = \mathit{compile}(\Gamma_i, \Gamma_m\ \mathit{mod}, \Gamma_r, \overrightarrow{\mathit{comp}}) \\
\quad \mathit{compile}(\Gamma_i, \Gamma_m, \Gamma_r, (\textbf{link}\ n\ (n_1\ n_2))\ \overrightarrow{\mathit{comp}}) \\
\qquad = \mathit{compile}(\Gamma_i, \Gamma_m\ \mathit{link}(\Gamma_i, n, \Gamma_m(n_1), \Gamma_m(n_2)), \Gamma_r, \overrightarrow{\mathit{comp}}) \\
\quad \mathit{compile}(\Gamma_i, \Gamma_m, \Gamma_r, (\textbf{invoke}\ n)\ \overrightarrow{\mathit{comp}}) \\
\qquad = \overrightarrow{\mathit{verify}(\Gamma_i, \mathit{body}_2)}\ \mathit{compile}(\Gamma_i, \Gamma_m, \Gamma_r\ \overrightarrow{re}, \overrightarrow{\mathit{comp}}) \\
\qquad\quad \text{where } \Gamma_m(n) = (\textbf{module}\ n\ \overrightarrow{\mathit{body}_1}) \\
\qquad\quad \text{and } \mathit{rename}(\overrightarrow{\mathit{body}_1}) = \overrightarrow{\mathit{body}_2} \\
\qquad\quad \text{and } \{\overrightarrow{re_0} \mid (\textbf{import}\ n\ \overrightarrow{re_0}) \in \overrightarrow{\mathit{body}_2} \text{ or } (\textbf{export}\ n\ \overrightarrow{re_0}) \in \overrightarrow{\mathit{body}_2}\} = \overrightarrow{re} \\
\quad \mathit{compile}(\Gamma_i, \Gamma_m, \overrightarrow{(f_1\ f_2)}, \mathit{expr}\ \overrightarrow{\mathit{comp}}) \qquad = \mathit{expr}[\overrightarrow{f_1 = f_2}]\ \mathit{compile}(\Gamma_i, \Gamma_m, \overrightarrow{(f_1\ f_2)}, \overrightarrow{\mathit{comp}}) \\
\quad \mathit{compile}(\Gamma_i, \Gamma_m, \Gamma_r, \epsilon) \qquad = \epsilon \\
\\
\mathit{link} : \Gamma_i, n, \mathit{mod}, \mathit{mod} \to \mathit{mod} \\
\quad \mathit{link}(\Gamma_i, n, (\textbf{module}\ n_1\ \overrightarrow{\mathit{body}_1}), \qquad = (\textbf{module}\ n\ \overrightarrow{\mathit{body}_3}\ \overrightarrow{\mathit{body}_4}) \\
\qquad\qquad (\textbf{module}\ n_2\ \overrightarrow{\mathit{body}_2})) \qquad\quad \text{where } \mathit{rename}(\overrightarrow{\mathit{body}_1}) = \overrightarrow{\mathit{body}_3} \\
\qquad\qquad\qquad\qquad\quad \text{and } \mathit{resolve}(\overrightarrow{\mathit{body}_3}, \mathit{rename}(\overrightarrow{\mathit{body}_2})) = \overrightarrow{\mathit{body}_4} \\
\\
\mathit{resolve} : \overrightarrow{\mathit{body}}, \overrightarrow{\mathit{body}} \to \overrightarrow{\mathit{body}} \\
\quad \mathit{resolve}(\overrightarrow{\mathit{body}_1}, \epsilon) \qquad = \epsilon \\
\quad \mathit{resolve}(\overrightarrow{\mathit{body}_1}, (\textbf{import}\ n\ \overrightarrow{(f\ f_1)})\ \overrightarrow{\mathit{body}_2}) \qquad = \mathit{resolve}(\overrightarrow{\mathit{body}_1}, \overrightarrow{\mathit{body}_2[\overrightarrow{f_1 = f_2}]}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{if } (\textbf{import}\ n\ \overrightarrow{(f\ f_2)}) \in \overrightarrow{\mathit{body}_1} \\
\quad \mathit{resolve}(\overrightarrow{\mathit{body}_1}, (\textbf{import}\ n\ \overrightarrow{(f\ f_1)})\ \overrightarrow{\mathit{body}_2}) \qquad = \mathit{resolve}(\overrightarrow{\mathit{body}_1}, \overrightarrow{\mathit{body}_2[\overrightarrow{f_1 = f_2}]}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{if } (\textbf{export}\ n\ \overrightarrow{(f\ f_2)}) \in \overrightarrow{\mathit{body}_1} \\
\quad \mathit{resolve}(\overrightarrow{\mathit{body}_1}, (\textbf{import}\ n\ \overrightarrow{re})\ \overrightarrow{\mathit{body}_2}) \qquad = (\textbf{import}\ n\ \overrightarrow{re})\ \mathit{resolve}(\overrightarrow{\mathit{body}_1}, \overrightarrow{\mathit{body}_2}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{if } n \notin \overrightarrow{\mathit{body}_1} \\
\quad \mathit{resolve}(\overrightarrow{\mathit{body}_1}, \mathit{defn}\ \overrightarrow{\mathit{body}_2}) \qquad = \mathit{defn}\ \mathit{resolve}(\overrightarrow{\mathit{body}_1}, \overrightarrow{\mathit{body}_2}) \\
\quad \mathit{resolve}(\overrightarrow{\mathit{body}_1}, \mathit{ex}\ \overrightarrow{\mathit{body}_2}) \qquad = \mathit{ex}\ \mathit{resolve}(\overrightarrow{\mathit{body}_1}, \overrightarrow{\mathit{body}_2}) \\
\\
\mathit{rename} : \overrightarrow{\mathit{body}} \to \overrightarrow{\mathit{body}} \\
\quad \mathit{rename}(\overrightarrow{\mathit{body}}) \qquad = \overrightarrow{\mathit{body}[\overrightarrow{f_1 = f_2}]} \\
\qquad\qquad\qquad\qquad\quad \text{where } \overrightarrow{\mathit{introduced}(\mathit{body})} = \overrightarrow{f_1}^n \\
\qquad\qquad\qquad\qquad\quad \text{and } \overrightarrow{f_2}^n \text{ fresh} \\
\\
\mathit{introduced} : \mathit{body} \to \overrightarrow{f} \\
\quad \mathit{introduced}((\textbf{import}\ n\ \overrightarrow{(f_1\ f_2)})) = \overrightarrow{f_2} \qquad \mathit{introduced}((\textbf{defstub}\ f\ (\overrightarrow{x})\ \mathtt{t})) = f \\
\quad \mathit{introduced}(ex) = \epsilon \qquad \mathit{introduced}((\textbf{defthm}\ f\ \mathit{expr})) = f \\
\quad \mathit{introduced}((\textbf{defun}\ f\ (\overrightarrow{x})\ \mathit{expr})) = f \qquad \mathit{introduced}((\textbf{skip-proofs}\ \mathit{defn})) = \mathit{introduced}(\mathit{defn})
\end{array}
$$

Figure 2.7: Translation from a Modular ACL2 program to an executable ACL2 program.

We extract executable definitions from invoked modules in the same way we extract proof obligations during verification (*verify*; see section 2.4.1). This produces each module's internal definitions, along with the contracts and manifest functions of their exported interfaces (as **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.

2.3.2 Execution Example

Once again, consider the finite set representation from figure 2.2. 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)))))
```

$$\textsc{pfmprog}\ \frac{\epsilon \vdash_c \mathit{mprog}}{\vdash_p \mathit{mprog}} \qquad \textsc{pfcomp0}\ \frac{}{\Gamma_i \vdash_c \epsilon} \qquad \textsc{pfcexpr}\ \frac{\Gamma_i \vdash_c \overrightarrow{\mathit{comp}}}{\Gamma_i \vdash_c \mathit{expr}\ \overrightarrow{\mathit{comp}}} \qquad \textsc{pfifc}\ \frac{\Gamma_i\ \mathit{ifc} \vdash_c \overrightarrow{\mathit{comp}}}{\Gamma_i \vdash_c \mathit{ifc}\ \overrightarrow{\mathit{comp}}}$$

$$\textsc{pfmod}\ \frac{\vdash_s \mathit{obligations}(\Gamma_i, \mathit{mod}) \qquad \Gamma_i \vdash_c \overrightarrow{\mathit{comp}}}{\Gamma_i \vdash_c \mathit{mod}\ \overrightarrow{\mathit{comp}}} \qquad \textsc{pflink}\ \frac{\Gamma_i \vdash_c \overrightarrow{\mathit{comp}}}{\Gamma_i \vdash_c \mathit{link}\ \overrightarrow{\mathit{comp}}} \qquad \textsc{pfinv}\ \frac{\Gamma_i \vdash_c \overrightarrow{\mathit{comp}}}{\Gamma_i \vdash_c \mathit{inv}\ \overrightarrow{\mathit{comp}}}$$

Figure 2.8: Inference rules for Modular ACL2 verification.

```
(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 2.1. 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. Once the atomic modules are verified, this program can be safely run in ACL2's program mode.

2.4 Verification

The logical semantics for Modular ACL2 turns a whole Modular ACL2 program into a series of ACL2 programs representing the proof obligations for each module in the source program. These proof obligations can be verified separately and in any order using ACL2. Once all of the proof obligations are verified, the executable form of the whole program is guaranteed to be logically valid as well.

2.4.1 Verifying Modules

We formalize the process of generating proof obligations for Modular ACL2 programs in figures 2.8 and 2.9. 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 2.9 defines substitution on (Modular) ACL2 terms.

The main judgment in the verification of modular programs is $\vdash_p \mathit{mprog}$, meaning that the program's components can be verified by ACL2. This is defined in terms of $\Gamma_i \vdash_c \overrightarrow{\mathit{comp}}$, meaning that the components can be verified in the context of additional interfaces, and

$$
\begin{array}{l}
\mathit{obligations} : \Gamma_i, \mathit{mod} \to \mathit{prog} \\
\quad \mathit{obligations}(\Gamma_i, (\textbf{module}\ n\ \overrightarrow{\mathit{body}})) \\
\quad = \overrightarrow{\mathit{verify}(\Gamma_i, \mathit{body})}
\end{array}
$$

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

$$
\begin{array}{l}
\mathit{assume} : \mathit{spec} \to \overrightarrow{\mathit{term}} \\
\quad \mathit{assume}((\textbf{fun}\ f\ (\overrightarrow{x})\ \mathit{expr})) = (\textbf{skip-proofs}\ (\textbf{defun}\ f\ (\overrightarrow{x})\ \mathit{expr})) \\
\quad \mathit{assume}((\textbf{sig}\ f\ (\overrightarrow{x}))) = (\textbf{defstub}\ f\ (\overrightarrow{x})\ \mathtt{t}) \\
\quad \mathit{assume}((\textbf{con}\ f\ \mathit{expr})) = (\textbf{skip-proofs}\ (\textbf{defthm}\ f\ \mathit{expr})) \\
\quad \mathit{assume}((\textbf{extend}\ n)) = \epsilon
\end{array}
$$

$$
\begin{array}{l}
\mathit{assert} : \mathit{spec} \to \overrightarrow{\mathit{term}} \\
\quad \mathit{assert}((\textbf{fun}\ f\ (\overrightarrow{x})\ \mathit{expr})) = (\textbf{defun}\ f\ (\overrightarrow{x})\ \mathit{expr}) \\
\quad \mathit{assert}((\textbf{sig}\ f\ (\overrightarrow{x}))) = \epsilon \\
\quad \mathit{assert}((\textbf{con}\ f\ \mathit{expr})) = (\textbf{defthm}\ f\ \mathit{expr}) \\
\quad \mathit{assert}((\textbf{extend}\ n)) = \epsilon
\end{array}
$$

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

Figure 2.9: Metafunctions for Modular ACL2 verification.

$\vdash_s \mathit{prog}$, meaning that ACL2 verifies the program. We defer the definition of this judgment until section 2.5.

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

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

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

Figure 2.10: Inference rules for ACL2 verification.

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

$\mathit{measure} : f, \overrightarrow{x}, expr \to expr$
termination conditions (omitted)

$\mathit{induction} : f, \overrightarrow{x}, expr \to expr$
induction schemes (omitted)

Figure 2.11: Metafunctions for ACL2 verification.

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.

2.4.2 Verification Example

To illustrate the verification process, we construct proof obligations for the set representation from figure 2.2. 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 2.1, 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))))
```

These definitions are all present in the original set representation of figure 2.1; 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.

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

Figure 2.12: Equational reasoning used in lemma 3.

2.5 Soundness

Adding linguistic machinery to the programming language of a theorem prover demands a rigorous soundness proof, which we provide in this section. 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 experiments from section 2.2 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.

2.5.1 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 2.10, is based on Kaufmann and Moore's work [Kaufmann and Moore 1998, 2001]. Supporting definitions are in figure 2.11.

The primary judgment, $\vdash_s prog$, defines the provability of whole programs. It is built up by iteration over terms. Analogously, the judgment $\Gamma_e \vdash_t \overrightarrow{term}$ 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 $\Gamma_e \vdash_d defn$. Then their conclusions are added to the environment. The judgment $\Gamma_e \vdash_e expr$ means that *expr* is provably valid and is used to check 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 [Kaufmann and Moore 1998] 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 [Kaufmann and Moore 1998] and the well-known properties of first-order logic.

2.5.2 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 2.10 (for ACL2) and 2.8 (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 brevity we do not present a formal treatment of them here.

We use the abbreviation $\Gamma_i \vdash_m \Gamma_m$ to mean $\Gamma_m = \overrightarrow{mod}$ implies $\vdash_s \overrightarrow{\mathit{obligations}(\Gamma_i, mod)}$.

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

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*
$\Gamma_e \vdash_t \mathit{compile}(\Gamma_i, \Gamma_m, \Gamma_r, \overrightarrow{comp})$.

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 Γ_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, we show that $\mathit{link}(\Gamma_i, n, \Gamma_m(n_1), \Gamma_m(n_2))$ 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. $\vdash_s$ *obligations*$(\Gamma_i, \mathit{link}(\Gamma_i, n, mod_1, mod_2))$ *holds if* $\vdash_s$ *obligations*(Γ_i, mod_1) *and* $\vdash_s$ *obligations*(Γ_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, *link*$(\Gamma_i, n, mod_1, mod_2)$ is (**module** n $\overrightarrow{body_3}$ $\overrightarrow{body_4}$) where $\overrightarrow{body_3} =$ *rename*$(\overrightarrow{body_1})$ and $\overrightarrow{body_4} =$ *resolve*$(\overrightarrow{body_3}, \mathit{rename}(\overrightarrow{body_2}))$. By rule PFPROG and the definition of *obligations*, we must prove:

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

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:

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

Lemma 6, applied to $\overrightarrow{body_2}$, shows that *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* $\Gamma_e \vdash_t \overrightarrow{\mathit{verify}(\Gamma_i, body_2)}$ *and*
$\overrightarrow{body_3} =$ *resolve*$(\overrightarrow{body_1}, \overrightarrow{body_2})$*, then*
$\Gamma_e\ \overrightarrow{\mathit{theory}(\mathit{verify}(\Gamma_i, body_1))} \vdash_t \mathit{verify}(\Gamma_i, \overrightarrow{body_3})$.

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 2.12.

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

Case $n \notin \overrightarrow{body_1}$**:** Trivial. □

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

Lemma 5. *verify*$(\Gamma_i, \overrightarrow{body})[\overrightarrow{f_1 = f_2}] =$ *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* $\Gamma_e \vdash_t$ *rename*$(\overrightarrow{defn})$.

Lemma 7. *If* $\Gamma_e \vdash_t \overrightarrow{term}$ *and* $f \notin \overrightarrow{term}$*, then* $\Gamma_e \vdash_t \overrightarrow{term}[\overrightarrow{f_0 = f}]$.

Proof. This follows from the proof of soundness of simple functional instantiations [Kaufmann and Moore 2001]. 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* $\Gamma_e\ \overrightarrow{\textit{theory}(defn_1)} \vdash_t \overrightarrow{defn_2}$

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

Lemma 9. *theory*$($*assume*$(spec)) =$ *theory*$($*assert*$(spec))$.

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

2.5.3 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 module 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.

Chapter 3

Hygienic Macros

Hygienic ACL2[1] adapts the hygienic, scope-respecting macro system of Scheme to ACL2, supplanting the unhygienic macro system it inherits from Common Lisp. We present the high-level design and low-level semantics of Hygienic ACL2, along with an evaluation of the impact of hygiene on existing ACL2 macros.

3.1 Motivation

From Common Lisp, ACL2 inherits *macros*, which provide a mechanism for extending the language via functions that operate on syntax trees. According to Kaufmann and Moore [1994], "*one can make specifications more succinct and easy to grasp ... by introducing well-designed application-specific notation.*" Indeed, macros are used ubiquitously in ACL2 libraries: there are macros for pattern matching; for establishing new homogenous list types and heterogenous structure types, including a comprehensive theory of each; for defining quantified claims using skolemization in an otherwise (explicit) quantifier-free logic; and so on.

In the first-order language of ACL2, macros are also used to eliminate repeated syntactic patterns due to the lack of higher-order functions:

```
(defmacro defun-map (map-fun fun)
  `(defun ,map-fun (xs)
     (if (endp xs)
       nil
       (cons (,fun (car xs)) (,map-fun (cdr xs))))))
```

This macro definition captures the essence of defining one function that applies another pointwise to a list. It consumes two inputs, map-fun and fun, representing function names; the body constructs a suitable **defun** form. ACL2 *expands* uses of defun-map, supplying

[1] This work has been published as Eastlund and Felleisen [2010].

the syntax of its arguments as map-fun and fun, and continues with the resulting function definition. Consider the following terms:

```
(defun double (x) (+ x x))

(defun-map map-double double)
```

Their expansion fills the names map-double and double into defun-map's template. The resulting second definition is:

```
(defun map-double (xs)
  (if (endp xs)
    nil
    (cons (double (car xs)) (map-double (cdr xs)))))
```

Unfortunately, ACL2 macros are *unhygienic* [Kohlbecker et al. 1986], meaning they do not preserve the meaning of variable bindings and references during code expansion. The end result is accidental capture that not only violates a programmer's intuition of lexical scope but also interferes with logical reasoning about the program source. In short, macros do not properly abstract over syntax. For instance, consider the or macro, which encodes both boolean disjunction and recovery from exceptional conditions, returning the second value if the first is nil:

```
(defthm excluded-middle (or (not x) x))

(defun find (n xs) (or (nth n xs) 0))
```

The first definition states the law of the excluded middle. Since ACL2 is based on classical logic, either (not x) or x must be true for any x. The second defines selection from a list of numbers: produce the element of xs at index n, or return 0 if nth returns nil, indicating that the index is out of range.

A natural definition for or duplicates its first operand:

```
(defmacro or (a b) `(if ,a ,a ,b))                    (3.1)
```

This works well for excluded-middle, but the expanded version of find now traverses its input twice, doubling its running time:

```
(defun find (n xs) (if (nth n xs) (nth n xs) 0))
```

Macro users should not have to give up reasoning about their function's running time. Consequently, macros should avoid this kind of code duplication.

The next logical step in the development of or saves the result of its first operand in a temporary variable:

```
(defmacro or (a b) `(let ((x ,a)) (if x x ,b)))       (3.2)
```

This macro now produces correct and efficient code for find. Sadly though, the expanded form of excluded-middle is no longer the expected logical statement:

(**defthm** excluded-middle (let ((x (not x))) (if x x x)))

Specifically, the or macro's variable x has captured excluded-middle's second reference to x. As a result, the conjecture is now equivalent to the unqualified statement (not x).

ACL2 resolves this issue by providing two different behaviors for the or macro. For symbolic verification, or expands using code duplication. For execution, it expands by introducing a fresh variable that cannot capture any other. This kind of expansion only works for or as a special case built in to the ACL2 compiler. Ordinary macros are not allowed to use separate expansion for verification and execution, as this is an invitation to unsoundness. Users' macros also cannot introduce fresh variables, as this is a side effect that has no logical foundation.

In general, macro writers tread a fine line. Many macros duplicate code to avoid introducing a variable that might capture bindings in the source code. Others introduce esoteric temporary names to avoid accidental capture. None of these solutions is universal, and thus the Scheme community introduced the notion of *hygienic* macros [Kohlbecker et al. 1986; Clinger and Rees 1991; Dybvig et al. 1992]. This chapter presents an adaptation of hygienic macros to ACL2.[2] Next we motivate the design and the ACL2-specific challenges, followed by a sketch of the implementation, and finally a comprehensive evaluation of the system vis-a-vis the ACL2 code base.

3.2 Design

Hygienic macro systems enforce a meaning for variables in macro-generated code based on the lexical scope policies of the host language. In order to develop our system, we must first analyze lexical scope in ACL2 and decide what meanings are appropriate for macro-generated references and bindings.

3.2.1 Design Goals

Our hygienic macro expander is designed to observe three key principles.

Referential transparency means that variables derive their meaning from where they occur and retain that meaning throughout the macro expansion process. Specifically, variable references inserted by a macro refer to bindings inserted by the macro or to bindings apparent at its definition site. Symmetrically, variable references in macro arguments refer to

[2]The only other theorem prover with hygienic macros is NUPRL Griffin [1988]; however, its macro system is much more simplistic than ACL2's.

d	$\in$	def	=	(mutual-recursion $\overrightarrow{(\textbf{defun}\ sym\ (\overrightarrow{sym})\ exp)}$) mutually recursive functions
			|	(**defmacro** sym $(\overrightarrow{sym})$ exp) macro definition
			|	(**defthm** sym exp $\overrightarrow{(sym\ \overrightarrow{(sym\ exp)})}$) conjecture with proof hints
			|	(include-book str) library import
			|	(encapsulate $(\overrightarrow{(sym\ num)})$ $\overrightarrow{def}$) definition block
			|	(local def) local definition
e	$\in$	exp	=	sym variable reference
			|	$(sym\ \overrightarrow{exp})$ function call
			|	(let $(\overrightarrow{(sym\ exp)})$ exp) lexical value bindings
			|	(**quote** $term$) literal value

Figure 3.1: Syntax of fully-expanded ACL2 programs.

bindings apparent at the macro call site. In the tradition of prior hygienic macro systems, we provide a disciplined method for manually violating this principle when needed.

Next, *separate compilation* for libraries demands that their compiled form may still be loaded into a running program. There is no need to re-expand the contents of a library each time it is included in a new context.

Finally, *source compatibility* means that our changes to ACL2 have negligible impact on working programs. In particular, the evaluator and the verification system remain the same. The only observable difference in our system is the expansion of macros. Most well-behaved macros continue to function as before; changes in expansion suggest potentially flawed macros.

3.2.2 Reinterpreting ACL2

Figure 3.1 specifies a simplified grammar for ACL2; a program is a sequence of definitions. In source code, any definition or expression may be replaced by a macro application. The grammar is written in terms of symbols (sym), strings (str), numbers (num), and terms ($term$). We use this grammar to explain ACL2-specific challenges to hygienic macro expansion.

Lexical Bindings: ACL2 inherits Common Lisp's namespaces: function and variable bindings are separate and cannot shadow each other. The position of a variable reference determines its role. In an expression position, a variable refers to a value, in application position to a function or macro. For example, the following code uses both kinds of bindings for car:

```
(let ((car (car x))) (car car))
```

Hygienic expansion must track both function and variable bindings for each possible reference. After all, during expansion, the role of a symbol is unknown until its final position in the expanded code is determined.

Hygienic expansion must also be able to distinguish macro-inserted lexical bindings from those in source code or in other macros. For instance, with hygienic expansion, the version of or with a temporary variable (3.2) should work. That is, the excluded-middle conjecture should expand as follows:

(**defthm** excluded-middle (let ((x_m (not x))) (if x_m x_m x)))

The macro expander differentiates between the source program's x and the macro's x_m, as noted by the subscript; the conjecture's meaning is preserved.

Quantification: ACL2 conjectures are implicitly universally quantified:

;; claim: $\forall x(x > 0 \Rightarrow x \geq 0)$
(**defthm** non-negative (implies (> x 0) (>= x 0)))

Here the variable x is never explicitly bound, but its scope is the body of the **defthm** form. ACL2 discovers free variables during the expansion of conjectures and treats them as if they were bound.

This raises a question of how to treat free variables inserted by macros into conjectures. Consider the following definitions:

(**defmacro imply** (var) '(implies x ,var))

(**defthm** x=>x (**imply** x))

The body of x=>x expands into (implies x_m x), with x from x=>x and x_m from **imply**. In x=>x, x is clearly quantified by **defthm**. In the template of **imply**, however, there is no apparent binding for x. Therefore, the corresponding variable x_m in the expanded code must be considered unbound. Incorporating this interpretation into our semantics for hygienic expansion yields a new design rule: macros must not insert new free variables into conjectures.

We must not overuse this rule, however, as illustrated by the macro below:

(**defmacro disprove** (name body) '(defthm name (not ,body)))

(**disprove** x=x+1 (= x (+ x 1)))

Here we must decide what the apparent binding of x is in the body of x=x+1. In the source syntax, there is nothing explicit to suggest that x is a bound or quantified variable, but during expansion, the macro **disprove** inserts a **defthm** form that "captures" x and quantifies over it. On one hand, allowing this kind of capture violates referential transparency. On the

other hand, disallowing it prevents abstraction over **defthm**, because of the lack of explicit quantification.

We resolve the dilemma by allowing **defthm** to quantify over variables from a single source—surface syntax or a macro—but not over multiple variables from different sources. This permits macros that expand into **defthm**, which are common, but rejects accidental quantification, a source of bugs in the ACL2 code base.

Definition Scope: ACL2 performs macro expansion, verification, and compilation on one definition at a time. Forward references are disallowed, and no definition may overwrite an existing binding.

Nevertheless, just as hygiene prevents lexical bindings from different sources from shadowing each other, it also prevents definitions from different sources from overwriting each other. Consider the following two similar programs:

```
(defun f (x) (+ x 1))

(defmacro m () `(defun f (x) x))
(m)
```

```
(defmacro m () `(defun f (x) x))
(m)

(defun f (x) (+ x 1))
```

Both define a function f, and both define and invoke a macro **m** that defines a different function, also called f. In the left program, the top-level f is defined first. When **m** is defined, a binding for f is already apparent; the body of **m** uses the name of an existing binding. Therefore, the invocation of **m** constructs a definition that overwrites an existing binding, which is illegal.

In the right program, the macro **m** is defined before the top-level f. The invocation of **m** may therefore define f. Since this name originated inside the macro, its binding is not apparent to the rest of the program. Hence, the author of (the top-level) f has seen no prior definition for f, and should be free to choose this name. Hygienic expansion permits this program.

Our hygiene policy for definition names is thus two-fold. First, definition names inserted by macros—rather than taken from their input—are not made visible in the caller's context. Second, no definition may overwrite a name that already has a visible binding in the definition's context.

Encapsulated Abstractions: The encapsulate form in ACL2 delimits a block of definitions. Definitions are exported by default; these definitions represent the block's *constraint*, describing its logical guarantees to the outside. Definitions marked local represent a *witness* that can be used to verify the constraint, but they are not exported.

For example, the following block exports a constraint stating that $1 \leq 1$:

```
(encapsulate ()
  (local (defthm x<=x (<= x x)))
  (defthm 1<=1
    (<= 1 1) ;; use the following hint:
    (x<=x (x 1))))
```

The local conjecture states that (<= x x) holds for all values of x. The conjecture 1<=1 states that (<= 1 1) holds; the subsequent hint tells ACL2 that the previously verified theorem x<=x is helpful, with 1 substituted for x.

Once the definitions in the body of an encapsulate block have been verified, ACL2 discards hints and local definitions (the witness) and recompiles the remaining definitions (the constraint) in a second pass. The end result is a set of exported logical rules with no reference to the witness. Local theorems may not be used in subsequent hints, local functions and local macros may no longer be applied, and local names are available for redefinition.

An encapsulate block may have a third component, which is a set of *constrained functions*. The header of the encapsulate form lists names and arities of functions defined locally within the block. These functions' names are exported as part of the block's constraint; their definitions are not exported and remain part of the witness.

The following block exports a function of two arguments whose witness performs addition, but whose constraint guarantees only commutativity:

```
(encapsulate ((f 2))
  (local (defun f (x y) (+ x y)))
  (defthm commutativity (equal (f x y) (f y x))))
```

Definitions following this block can refer to f and reason about it as a commutative function. Attempts to prove it equivalent to addition would fail, however, and attempts to call it would result in a run-time error.

Our hygienic macro system preserves the scoping rules of encapsulate blocks. Furthermore, it enforces that names defined in the witness are not visible in the constraint, ensuring that a syntactically valid encapsulate block has a syntactically valid constraint. Our guarantee of referential transparency also means that local names in exported macros cannot be captured. For instance, the following macro **m** constructs a reference to w:

```
(encapsulate ()
  (local (defun w (x) x))
  (defmacro m (y) `(w ,y)))

(defun w (z) (m z)) ;; body expands to: (w z)
```

When a new w is defined outside the block and **m** is applied, the new binding does not capture the w from **m**. Instead, the macro expander signals a syntax error, because the inserted reference is no longer in scope.

Books: A *book* is the unit of ACL2 libraries: a set of definitions that is verified and compiled once and then reused. Each book acts as an encapsulate block without constrained functions; it is run twice—once with witness, and once for the constraint—and the constraint is saved to disk in compiled form. When a program includes a book, ACL2 incorporates its definitions, after ensuring that they do not clash with any existing bindings.

ACL2 allows an exception to the rule against redefinition that facilitates compatibility between books. Specifically, a definition is considered *redundant* and skipped, rather than rejected, if it is precisely the same as an existing one. If two books contain the same definition for a function f, for instance, the books are still compatible. Similarly, if one book is included twice in the same program, the second inclusion is considered redundant.

This notion of redundancy is complicated by hygienic macro expansion. Because hygienic expanders generally rename variables in their output, there is no guarantee that identical source syntax expands to an identical compiled form. As a result, redundancy becomes a question of α-equivalence instead of simple syntactic equality, and coalescing redundant definitions in compiled books requires renaming all references in addition to merely removing the second definition.

Rather than address redundancy in its full generality, we restrict it to the case of loading the same book twice. If a book is loaded twice, the new definitions will be syntactically equal to the old ones because books are only compiled once. That is, this important case of redundancy does not rely on α-equivalence.

Macros: Macros use a representation of syntax as their input and output. In the existing ACL2 system, syntax is represented using primitive data types: strings and numbers for literals, symbols for variables, and lists for sequences of terms.

In contrast, Dybvig et al. [1992] introduce a separate class of *syntax objects*: terms annotated with details of lexical scope and macro expansion. In order to preserve compatibility with existing ACL2 macros, we cannot introduce an entirely new data type. Instead, we adapt the method of Kohlbecker et al. [1986] by incorporating the annotations of syntax objects into ACL2's symbols.

In adapting the symbol datatype, we must be sure to preserve the axioms of ACL2; otherwise we risk invalidating existing proofs. On one hand, it is an axiom that any symbol is uniquely distinguished by the combination of its name and its *package*—an additional string used for manual namespace management. On the other hand, the hygienic macro expander must distinguish between symbols sharing a name and a package when one originates in the source program and another is inserted by a macro. We resolve this issue by leaving hygienic expansion metadata transparent to axiomatic primitives: only unverified functions such as macro definitions or the compiler itself can distinguish between two symbols with

the same name and package. Verified functions and theorems cannot make this distinction, i.e., ACL2's axioms remain valid.

The symbols inserted by macros must derive their lexical bindings from the context in which they appear. To understand the complexity of this statement, consider the following example:

```
(defun parse-compose (funs arg)
  (if (endp funs)
    arg
    `(,(car funs) (compose ,(cdr funs) ,arg))))

(defmacro compose (funs arg) (parse-compose funs arg))

(compose (string-downcase symbol-name) 'SYM)
;; ⇒ (string-downcase (compose (symbol-name) 'SYM))
;; ⇒ (string-downcase (symbol-name (compose () 'SYM)))
;; ⇒ (string-downcase (symbol-name 'SYM))
```

The auxiliary function parse-compose creates recursive references to **compose**, but **compose** is not in scope in parse-compose. To support this common macro idiom, we give the code inserted by macros the context of the macro's definition site. In the above example, the symbol compose in parse-compose's template does not carry any context until it is returned from the **compose** macro, at which point it inherits a binding for the name. This behavior allows recursive macros with helper functions, at some cost to referential transparency: the reference inserted by parse-compose might be given a different meaning if used by another macro.

This quirk of our design could be alleviated if these macros were rewritten in a different style. If the helper function parse-compose accepted the recursive reference to **compose** as an argument, then the quoted symbol 'compose could be passed in from the definition of **compose** itself, where it has meaning.

```
(defun parse-compose (compose funs arg)
  (if (endp funs)
    arg
    `(,(car funs) (,compose ,(cdr funs) ,arg))))

(defmacro compose (funs arg) (parse-compose 'compose funs arg))
```

Symbols in macro templates could then take their context from their original position, observing referential transparency. However, because ACL2 macros are not generally written in this style, our design does not mandate it.

Breaking Hygiene: There are some cases where a macro must insert variables that do not inherit the context of the macro definition, but instead intentionally capture—or are

t	$\in$	$term$	$=$	$num \mid str \mid id \mid \mathsf{cons}(term, term)$							s-expression
i	$\in$	id	$=$	$sym \mid \mathsf{id}(sym, mset, ren, ren)$							identifier
x	$\in$	sym	$=$	$\mathsf{sym}(str, str, mset)$							symbol
b	$\in$	$bool$	$=$	$\mathbf{t} \mid \mathbf{nil}$							boolean
r	$\in$	ren	$=$	$[\overrightarrow{key \mapsto sym}]$	renaming	$\hat{x}$	$\in$	$xset$	$=$	$\{\overrightarrow{sym}\}$	symbol set
k	$\in$	key	$=$	$\langle sym, mset \rangle$	identifier key	$\hat{k}$	$\in$	$kset$	$=$	$\{\overrightarrow{key}\}$	key set
m	$\in$	$mark$	$=$	$\langle str, num \rangle$	mark	$\hat{m}$	$\in$	$mset$	$=$	$\{\overrightarrow{mark}\}$	mark set

Figure 3.2: Representation of values and syntax.

captured by—variables in the source program. For instance, the defun-map example can be rewritten to automatically construct the name of the map function from the name of the pointwise function:

```
(defmacro defun-map (fun)
  (let ((map-fun-string (string-append "map-" (symbol-name fun))))
    (let ((map-fun (in-package-of map-fun-string fun)))
      `(defun ,map-fun (xs)
         (if (endp xs)
           nil
           (cons (,fun (car xs)) (,map-fn (cdr xs))))))))

(defun-map double) ;; expands to: (defun map-double (xs) ...)
```

In this macro, the name double comes from the macro caller's context, but map-double is inserted by the macro itself. The macro's intention is to bind map-double in the caller's context, and the caller expects this name to be bound.

This implementation pattern derives from the Common Lisp package system. Specifically, the in-package-of function builds a new symbol with the given string as its name, and the package of the given symbol. In our example, map-double is defined in the same package as double.

We co-opt the same pattern to transfer lexical context. Thus map-double shares double's context, and its binding is visible to macro's caller. In general, macro writers can use in-package-of to break the default policy of hygiene.

3.3 Semantics

Figure 3.2 describes the new representation of terms for hygienic macro expansion. The most important difference to a conventional representation concerns *identifiers*, which extend symbols to include information about expansion. Specifically, a term t is either a number, a string, an identifier, or a pair of terms. Numbers ($n \in num$) and strings ($s \in str$) are

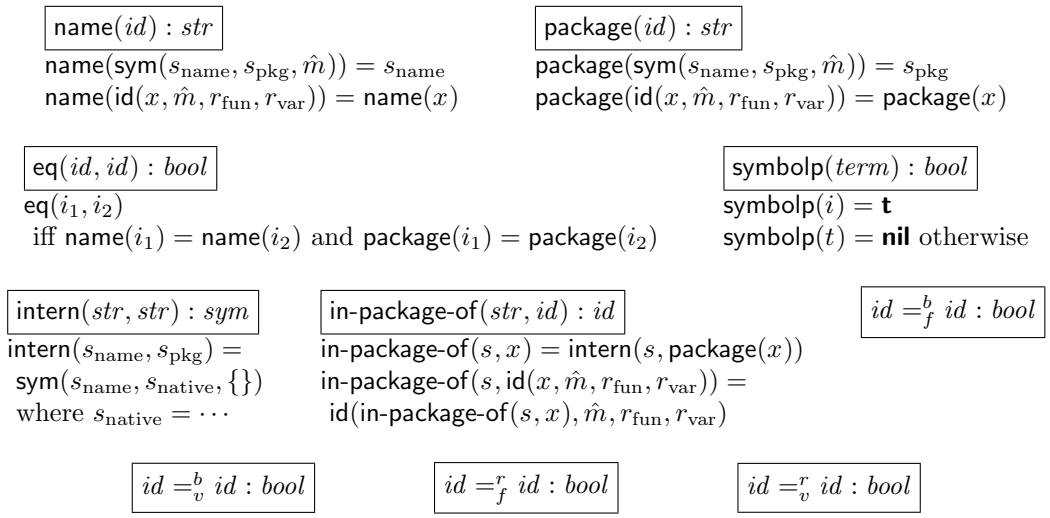

Figure 3.3: Updated ACL2 operations.

unchanged. An identifier i is either a symbol or an annotated symbol. A symbol x has three components: its name, its package, and a set of *inherent marks* used to support unique symbol generation. Annotated symbols contain a symbol, a set of *latent marks* used to record macro expansion steps, and two renamings assigning unique names to function and value bindings. We represent booleans with symbols, abbreviated **t** and **nil**.

Identifiers are used to represent variable names in unexpanded programs; unique symbol names are chosen for variables in fully expanded programs. The mapping between the two is mediated by *keys* (k); each function or value binding's key combines the symbol corresponding to the name's previous binding and the (latent) marks of the binding identifier. A *renaming* (r) maps surface names—keys—to compiled names—symbols.

A *mark* (m) is used to uniquely identify an event in macro expansion. Each one comprises its source file as a string—to distinguish marks generated during the compilation of separate books, in an adaptation of Flatt's mechanism for differentiating bindings from separate modules Flatt [2002]—as well as a number chosen uniquely during a single compilation session.

This representation of terms is used both for syntax during macro expansion and for values during ordinary runtime computation. Hence, ACL2 functions that deal with symbols must be updated to work with identifiers. The updated definitions of these operations are shown in figure 3.3.

The name and package functions produce the respective fields of a symbol, ignoring any identifier metadata. There is no explicit accessor for a symbol's inherent marks. Similarly, the equality predicate eq ignores identifier metadata and inherent marks. Two identifiers

ψ	$\in$	*state*	$=$	$\langle str, bool, bool, ren, table, xset, kset\rangle$	expansion state
τ	$\in$	*table*	$=$	$[\overrightarrow{sym \mapsto rec}]$	def. table
ρ	$\in$	*rec*	$=$	$\langle sig, fun, thm\rangle$	def. record
σ	$\in$	*sig*	$=$	$\mathsf{fun}(bool, num) \mid \mathsf{macro}(id, num) \mid \mathsf{thm}(xset) \mid \mathsf{special}$	def. signature
ϕ	$\in$	fun^n	$=$	$\cdot \mid \overrightarrow{term}^n \to term$	n-ary function
θ	$\in$	*thm*	$=$	$\cdot \mid \cdots$	theorem formula

Figure 3.4: Representation of expansion state.

are considered equal by the ACL2 logic and runtime if their observable fields are the same.

The intern function produces symbols from a name and package, assigning them an empty inherent mark set. We elide the details by which symbol names are looked up in a package table to find a symbol's "native" package. The related function in-package-of takes a string and an identifier, and constructs a new identifier with the string as its name, the package and metadata of the given identifier, and no inherent marks. Essentially, $\mathsf{in\text{-}package\text{-}of}(s, i)$ represents the variable name s in the lexical context of i.

We also introduce four new identifier comparisons: $=_f^b$, $=_f^r$, $=_v^b$, and $=_v^r$. They are separated according to the ACL2 function and value namespaces, as signified by the subscripts, and to compare either references or binding occurrences, as signified by the superscripts. *These procedures do not respect ACL2's axioms.* They can distinguish between symbols with the same name, so we provide them as low-level Common LISP functions that cannot be accessed by ACL2's logic. Specifically, they may be used in macros as variable comparisons that respect apparent bindings.

Macro expansion requires an additional set of data representations to keep track of the expansion process. Figure 3.4 presents the relevant definitions. An expansion state ψ contains seven fields. The first names the source file being expanded. The second and third determine expansion modes: global versus local definition scope and "logic mode" versus "program mode". Fields four and five provide mappings on the set of compiled definitions; one renames the keys of definitions to unique symbols, and the other maps those symbols to meanings. The sixth field is the subset of symbolic definition names that are exported from the nearest enclosing block, and the seventh is the set of keys for constrained functions that are declared but not yet defined in the current encapsulate block.

A definition table τ maps each definition to a record ρ describing its signature, executable behavior, and logical meaning. A signature σ describes the various kinds of definitions: functions of a particular mode ("logic" or "program") and arity, macros of a given lexical context and arity, theorems with a set of quantified variables, and special built-in definitions like **defun** and encapsulate. An executable function ϕ implements a function or macro, and

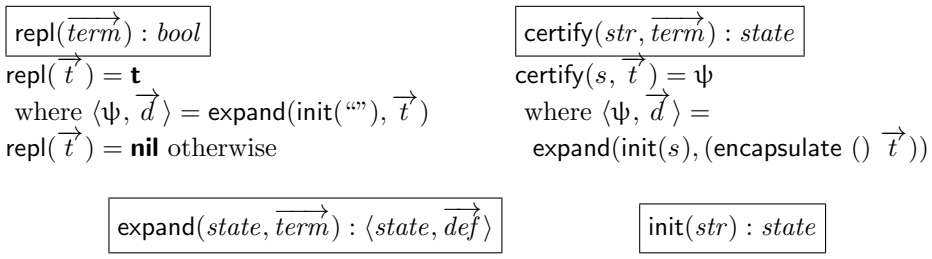

Figure 3.5: Top-level expansion.

a logical formula θ describes a function or theorem equationally.

There are two entry points into the ACL2 expansion, verification, and execution process: repl, which manages interactive verification, and certify, which verifies saved books and records their compiled form. Both are defined in figure 3.5 in terms of the underlying function expand.

Interactive programs consist of a sequence of unexpanded definitions. Expansion proceeds from an initial state constructed by init, not shown here. Interactive verification reports whether a program is syntactically and logically valid.

Certification of books requires the name of the book and the sequence of terms within it. The process begins with the same initial state as repl except for the source name used to generate unique symbols. The terms in the book are wrapped in encapsulate to protect local definitions. Certification produces a compiler state whose definitions can be merged into future programs.

The expand function is the workhorse of the macro expander. It expands definitions in a sequence, whether from the user, a book, or encapsulate, producing their expanded form along with an updated expansion state. It dispatches to constraint to build the constraint for books and encapsulate, discarding the witness, and to expr to expand expressions in functions, conjectures, and macros.

Our expander relies on the existing behavior of verification and compilation. Once the defn and constraint functions obtain fully expanded code, they dispatch to the underlying compiler and proof engine for ACL2. In this way, we preserve the semantics of all stages beyond macro expansion.

3.4 Evaluation

Our design goals for hygienic ACL2 macros mention three guiding principles: referential transparency, separate compilation, and source compatibility. As explained, the macro ex-

pansion process preserves referential transparency by tracking the provenance of identifiers, with two key exceptions: symbols inserted by macros take their context from the macro definition site rather than their own occurrence, and conjecture quantification can "capture" free variables in macro inputs. Furthermore, our representation for compiled books guarantees separate compilation. Only source compatibility remains to be evaluated.

Our preliminary prototype does not support many of the advanced, non-macro-related features of ACL2 and we are thus unable to run hygienic expansion on most existing books. To determine the degree of compatibility between our system and existing macros, we manually inspected all 2,954 **defmacro** forms in the books provided with ACL2 version 3.6, including the separate package of books accumulated from the ACL2 workshop series. Most of these macros are simple aliases for previously defined functions and would not be affected one way or another by the introduction of hygiene. We found 488 nontrivial macros that interact with hygienic expansion, which we discuss in this section.

Code Duplication: The behavior of macro-duplicated code does not change with hygienic expansion; however, hygiene encourages the introduction of local variables in macros and thus avoids duplication. With our system, all 130 code-duplicating macros can benefit from hygiene.

Variable Comparison: Comparing variable names with `eq` does not take into account their provenance in the macro expansion process and can mistakenly identify two symbols with the same name but different lexical contexts. We found 26 macros in ACL2 that compare variable names for assorted purposes, none of which are served if the comparison does not respect the variable's binding. The new functions $=_f^b$, $=_f^r$, $=_v^b$, and $=_v^r$ provide comparisons for variables that respect lexical context. Once again, the result of `eq` does not change in our system, so these macros will function as they have; however, macro writers now have the option of using improved tools.

Free Variables: Free variables in macros usually represent some protocol by which macros take their meaning from their context; i.e., they must be used in a context where the names in question have been bound. Much like mutable state in imperative languages, free variables in macros represent an invisible channel of communication. When used judiciously, they create succinct programs, but they can also be a barrier to understanding. Of the 90 cases of macros that insert free variables, 83 are based on such a protocol. Of these, 36 rely on built-in ACL2 features that bind names implicitly. Our hygienic macro expander will reject uses of these 83 macros; they must be rewritten to accept explicit arguments.

Five further cases of "free variables" are forward references, in which a macro's body constructs a reference to a subsequent function or macro. To a macro writer, this may not seem like a "free" reference, but it is, due to the scope of ACL2 definitions. Therefore this

use of forward references does not satisfy the principle of referential transparency. These macros must also be rewritten or reordered to function under hygienic macro expansion.

The final two cases of free variables in a macro were, in fact, bugs. The macro is used to generate the body of a conjecture. It takes several expressions and splices them into a large implication. One of the inputs is named top, and its first reference in the implication is accidentally quoted—that is, instead of filling in the contents of the input named top, the macro inserts a literal reference to a variable named top. By serendipity, this macro is passed a variable named top, and nothing goes wrong. Were this macro ever to be used with another name, it would construct the wrong conjecture and either fail due to a mysterious extra variable or succeed spuriously by proving the wrong proposition. Our interpretation of hygienic expansion for **defthm** catches this bug early, reporting a conflict between quantified variables from different contexts.

Variable Capture: We found 242 instances of variable (85) or definition (157) names inserted by macros that introduce bindings to the macro's input or surrounding program. Of the macros that insert definition names, there were 95 that used in-package-of to explicitly bind names in the package of their input, 44 that used intern to bind names in their own package, 16 that used hard-coded names not based on their input at all, and two that used the make-event facility Kaufmann and Moore [2009] to construct unique names.

The package-aware macros will continue to function as before. The intern-based macros guarantee neither that the constructed names bind in the context of the input, nor that they don't. In contrast, hygienic expansion provides a consistent guarantee that they don't, making their meaning predictable. Hard-coded names in macros will no longer bind outside of the macro itself. These are the other side of free variable protocols; they must be made explicit to interoperate with hygiene. The make-event utility allows inspection of the current bindings to construct a unique name, but nothing prevents that name from clashing with any subsequent binding. Hygiene eliminates the need to manually scan the current bindings and guarantees global uniqueness.

Local variables account for the other 85 introduced bindings. We discovered nine whose call sites exploited these bindings as part of an intentional protocol. These macros can be made hygienic by taking the variable name in question as an argument, thus making the macro compatible with hygienic expansion, freeing up a name the user might want for something else, and avoiding surprises if a user does not know the macro's protocol.

Of the other 76 macros that bind local variables in the scope of their arguments, 59 attempt to avoid capture. There are 12 that choose long, obscure names; for instance, gensym::metlist (meaning "metlist" in the "gensym" package), indicating a wish for the Lisp symbol-generating function gensym, which is not available in ACL2. There is also a

	Improves for free	Improves with work	Unchanged	Broken; improves	Broken; restores
Code Duplication	–	130	–	–	–
Free variable	2	–	–	83	5
Lexical capture	29	47	–	9	–
Definition capture	–	2	95	44	16
Variable comparison	–	26	–	–	–
Total	31	205	95	136	21

Figure 3.6: Impact of hygienic expansion on nontrivial ACL2 macros.

convention of adding -do-not-use-elsewhere or some similar suffix to macro-bound variables; in one case, due to code copying, a variable named hyp--dont-use-this-name-elsewhere is in fact bound by *two* macros in different files. Obscure names are a poor form of protection when they are chosen following a simple formula, and a macro that binds a hard-coded long name will never compose properly with itself, as it always binds the same name.

A further 40 macros generate non-capturing names based on a known set of free variables, and seven more fail with a compile error if they capture a name as detected by check-vars-not-free. These macros are guaranteed not to capture, but the latter still force the user to learn the name bound by the macro and avoid choosing it for another purpose. Some of these macros claim common names, such as val and x, for themselves.

Finally, we have found 17 macros in the ACL2 books that bind variables and take no steps to avoid capture. All of the accidentally variable-capturing macros will automatically benefit from hygienic expansion.

Exceptions: The notable exceptions to hygiene we have not addressed are make-event, a tool for relective code transformation, and state, a special variable used to represent mutation and i/o. We have not yet inspected most uses of make-event in the ACL2 code base, but do not anticipate any theoretical problems in adapting the feature. For state and similar "single-threaded" objects, our design must change so as to recognize the appropriate variables and not rename them.

Summary: Figure 3.6 summarizes our analysis. We categorize each macro by row according to the type of transformation it applies: code duplication, free variable insertion, capture of lexical or definition bindings, and variable comparison. We omit the trivial case of simple alias macros from this table.

We split the macros by column according to the anticipated result of hygienic expansion. In the leftmost column, we sum up the macros whose expansion is automatically improved by hygienic expansion. Next to that, we include macros that work as-is with hygiene, but permit a simpler definition. In the center, we tally the macros whose expansion is unaffected. To the right, we list macros that must be fixed to work with hygienic macro expansion, but

whose expansion becomes more predictable when fixed. In the rightmost column, we list those macros that must be fixed and gain no benefit from hygienic expansion.

Many libraries and built-in features of ACL2 rely on the unhygienic nature of expansion and use implicit bindings; as a result, our system cannot cope with every macro idiom in the code base. Fortunately, there appear to be few exceptions, and we anticipate that all of the macros distributed with ACL2 can be made to work with hygienic expansion, requiring at worst a straightforward fix to reorder definitions or accept explicit arguments in place of free variables. Furthermore, the bulk of macros will continue to work, and more will benefit from hygiene than will break in its presence. The frequent use of code duplication, obscure variable names, and other capture prevention mechanisms shows that ACL2 users recognize the need for a disciplined approach to avoiding unintentional capture in ACL2 macros.

Chapter 4

Extended Case Study

4.1 The Racket Virtual Machine

The Racket family of programming languages [Flatt and PLT 2010] operates by translating source code to a bytecode language, then executing the bytecode on a stack-based virtual machine (with a JIT compiler). The Racket virtual machine accepts an expression as input and submits it to a bytecode verifier. If the expression passes verification, it is translated into a machine state, which is then executed directly. The bytecode verifier ensures that a given expression performs only legal actions. To do this, it mimics the process of execution on an abstract machine that tracks the state of accessible locations for reading and writing values.

Our simplified model of a Racket machine state has three registers: value, code, and stack; it may alternately be in an error state. The value register is a *location*; the stack is a sequence of frames,[1] each of which is a sequence of locations. A location may be uninitialized or contain a value; machine values are closures with an arity, stored environment, and body expression.[2]

The code register contains a sequence of *instructions*, which may swap the value register with a stack location, set the value of a stack location, pop or push a set of two stack frames, call a function with some number of arguments, or evaluate a machine expression.

Our grammar for expressions encompasses the lambda calculus, modified to operate on a stack machine with multiple-argument functions. Variable lookup is performed by reference to stack locations. Each lambda abstraction carries an explicit arity, the stack locations of all free variables in its body, and the body expression itself. Function application includes one expression for the function to apply and a sequence of argument expressions. See figure 4.1 for the full grammar. We use the notation $\overrightarrow{x}$ to represent a sequence of terms of the form x, or $\overrightarrow{x}^n$ to specify the length n of the sequence.

[1] Our ACL2 implementation requires a stack to include at least one frame.

[2] Our ACL2 implementation also stores metadata for verification with each location; the model we

$$
\begin{array}{rcll}
m \in \mathit{mstate} & = & (\mathit{loc}, \mathit{stack}, \mathit{code}) \mid \mathsf{error} & \text{machine state} \\
s \in \mathit{stack} & = & \overrightarrow{\mathit{frame}} & \text{stack} \\
f \in \mathit{frame} & = & \langle \overrightarrow{\mathit{loc}} \rangle & \text{stack frame} \\
\ell \in \mathit{loc} & = & \mathit{value} \mid \mathsf{uninit} & \text{value location} \\
v \in \mathit{value} & = & \mathsf{clos}(\mathit{nat}, \overrightarrow{\mathit{value}}, \mathit{expr}) & \text{value} \\
c \in \mathit{code} & = & \overrightarrow{\mathit{instr}} & \text{machine code} \\
i \in \mathit{instr} & = & \mathsf{pop} \mid \mathsf{push} \mid \mathsf{set} \mid \mathsf{swap} \mid \mathsf{call} \mid \mathit{expr} & \text{instruction} \\
e \in \mathit{expr} & = & \mathsf{loc}(\mathit{nat}) & \text{expression} \\
 & & \mid \mathsf{lam}(\mathit{nat}, \overrightarrow{\mathit{nat}}, \mathit{expr}) & \\
 & & \mid \mathsf{app}(\mathit{expr}, \overrightarrow{\mathit{expr}}) &
\end{array}
$$

Figure 4.1: Grammar for machine states

Execution proceeds according to the single-step relation $\longrightarrow$, shown in figure 4.2. A loc expression copies the contents of the appropriate stack index to the value register. A lam expression creates a closure in the value register with the same arity and body, and with the values of free variables extracted from the stack. An app expression expands into a sequence of instructions that perform the steps in a function application: evaluate the function and argument expressions, each with a fresh set of two stack frames; store the function value in the value register and the arguments in new locations on top of the stack; and finally, call the function.

The push and pop instructions add and remove, respectively, the top two frames of the stack. Every function call makes use of two stack frames: the first stores temporary results in the evaluation of function arguments and the second stores the lexical environment and arguments of the current function call.[3]

Executing a set or swap instruction copies the contents of the value register to the indicated stack location; swap instructions also copy the value at that stack location to the value register.

Finally, the call instruction installs the body of a closure as the next instruction to execute and repopulates the top two stack frames for the new function call. If the instruction's arity does not match the arity of the closure in the value register, the call instruction instead produces an error state.

We omit full definitions of the straightforward stack manipulation functions, but we present their signatures in figure 4.3. The function stack-size reports the number of value locations in a stack. The pair of functions stack-ref and stack-set perform stack lookup and update based on location indices. Pairs of stack frames are added and removed by stack-push and stack-pop, and stack-alloc adds a sequence of locations to the top frame of a stack.

---

present here infers this metadata prior to verification.

[3]Our ACL2 model uses three stack frames, separating the lexical environment and function arguments.

$\boxed{mstate \longrightarrow mstate}$

$$
\begin{array}{lll}
(\ell, s, \mathsf{loc}(n)\ c) & \longrightarrow & (\mathsf{stack\text{-}ref}(s, n), s, c) \\
(\ell, s, \mathsf{lam}(n_{arity}, \overrightarrow{n_{free}}, e)\ c) & \longrightarrow & (\mathsf{clos}(n_{arity}, \overrightarrow{\mathsf{stack\text{-}ref}(s, n_{free})}, e), s, c) \\
(\ell, s, \mathsf{app}(e, \epsilon)\ c) & \longrightarrow & (\ell, s, \mathsf{push}\ e\ \mathsf{pop}\ \mathsf{call}(0)\ c) \\
(\ell, s, \mathsf{app}(e_{fun}, \overrightarrow{e_{arg}}^{n})\ c) & \longrightarrow & (\ell, s_{app}, c_{app}\ c) \\
 & & \quad \text{where} \\
 & & \quad\ s_{app} = \mathsf{stack\text{-}alloc}(\overrightarrow{\mathsf{uninit}}^{n}, s) \\
 & & \quad\ c_{app} = \mathsf{push}\ e_{fun}\ \mathsf{pop}\ \mathsf{set}(n-1)\ \mathsf{argc}(0, \overrightarrow{e_{arg}}) \\
(\ell, s, \mathsf{push}\ c) & \longrightarrow & (\ell, \mathsf{stack\text{-}push}(s), c) \\
(\ell, s, \mathsf{pop}\ c) & \longrightarrow & (\ell, \mathsf{stack\text{-}pop}(s), c) \\
(\ell, s, \mathsf{set}(n)\ c) & \longrightarrow & (\ell, \mathsf{stack\text{-}set}(s, n, \ell), c) \\
(\ell, s, \mathsf{swap}(n)\ c) & \longrightarrow & (\mathsf{stack\text{-}ref}(s, n), \mathsf{stack\text{-}set}(s, n, \ell), c) \\
(\ell, s, \mathsf{call}(n)\ c) & \longrightarrow & (\ell, \langle\rangle\ \langle \overrightarrow{v_{arg}}\ \overrightarrow{v_{free}} \rangle\ s_{rest}, e\ c) \\
 & & \quad \text{where} \\
 & & \quad\ \ell = \mathsf{clos}(n, \overrightarrow{v_{free}}, e) \\
 & & \quad\ s = \langle \overrightarrow{v_{arg}}^{n}\ \overrightarrow{\ell_{temp}} \rangle\ f_{env}\ s_{rest} \\
(\ell, s, \mathsf{call}(n)\ c) & \longrightarrow & \mathsf{error} \\
 & & \quad \text{where} \\
 & & \quad\ \ell = \mathsf{clos}(n_{arity}, \overrightarrow{v_{free}}, e) \\
 & & \quad\ n \neq n_{arity}
\end{array}
$$

$\boxed{\mathsf{argc}(nat, \overrightarrow{expr}) : code}$

$$
\begin{array}{lll}
\mathsf{argc}(n, e) & = & \mathsf{push}\ e\ \mathsf{pop}\ \mathsf{swap}(n)\ \mathsf{call}(n+1) \\
\mathsf{argc}(n, e_0\ \overrightarrow{e}) & = & \mathsf{push}\ e\ \mathsf{pop}\ \mathsf{set}(n)\ \mathsf{argc}(n+1, \overrightarrow{e})
\end{array}
$$

Figure 4.2: Execution of machine states

$$
\begin{array}{l}
\mathsf{stack\text{-}size}(stack) : nat \\
\mathsf{stack\text{-}ref}(stack, nat) : loc \\
\mathsf{stack\text{-}set}(stack, nat, loc) : stack \\
\mathsf{stack\text{-}push}(stack) : stack \\
\mathsf{stack\text{-}pop}(stack) : stack \\
\mathsf{stack\text{-}alloc}(\overrightarrow{loc}, stack) : stack
\end{array}
$$

Figure 4.3: Signatures for stack operations.

$$
\begin{array}{lll}
s^b \in bstack = \overrightarrow{bframe} \mid \mathsf{invalid} & & \text{bytecode abstract stack} \\
f^b \in bframe = \overrightarrow{bloc} & & \text{bytecode abstract stack frame} \\
\ell^b \in bloc = \mathsf{read} \mid \mathsf{neither} & & \text{bytecode abstract value location}
\end{array}
$$

Figure 4.4: Grammar for bytecode abstract stacks

For this language, bytecode verification amounts to ensuring that the stack locations in every loc and lam expression refer to positions that are guaranteed to be both allocated and initialized with a value by the time they are read. The bytecode verifier operates by simulating execution of an expression on a bytecode abstract stack that replaces the value at each location with an annotation of whether the location can be legally read or written. Abstract execution over-approximates control flow by checking all function bodies, even if they might not be executed at runtime.

A bytecode abstract stack for bytecode verification is represented as a sequence of bytecode abstract stack frames, or invalid to indicate that verification has failed. A bytecode abstract stack frame is a sequence of bytecode abstract values. Each bytecode abstract value in the stack is either a readable (but not writable) value, representing a variable in scope, or a location that is neither readable nor writable, representing space allocated for arguments to a function call. See figure 4.4 for the grammar of bytecode abstract stacks.

The bytecode verifier operates via the bverify function, shown in figure 4.5, which simulates a given expression starting with an empty stack via $\mathsf{bsimulate}_\mathsf{e}$ and ensures that the final stack is not invalid.

Within $\mathsf{bsimulate}_\mathsf{e}$, simulation of loc expressions amounts to checking whether the given stack index is allocated and readable. For lam expressions, the stack index of each free variable must be allocated and readable; furthermore, the body expression must be verified in the bytecode abstract stack corresponding to a function call: the body's free variables allocated on top of the function's arguments. An app expression is simulated by allocating an unreadable stack location for each argument, then simulating the operator expression and each argument in turn via $\mathsf{bsimulate}^*_\mathsf{e}$. The $\mathsf{bsimulate}^*_\mathsf{e}$ function simulates each expression in a new pair of stack frames, removing them before proceeding with the rest.

We generalize the stack operations from figure 4.3 to bytecode abstract stacks; invalid is considered to have stack-size of 0, and all other operations on invalid produce invalid.

Expressions are converted to machine states by first verifying them, then adding an uninitialized value register, a stack with two empty frames, and installing the expression as the sole initial instruction. This process is defined as init in figure 4.5.

4.2 Verifying the Verifier

Klein [2009] formulates the safety property stated for the bytecode verifier, adapted to our model, as follows:

> If the verifier accepts be and $\mathsf{init}(be) \longrightarrow^* (\ell, s^\ell, c)$, then either $c = \epsilon$ (i.e., no instructions remain) or $(\ell, s^\ell, c) \longrightarrow m$, for some machine state m.

$\boxed{\mathsf{init}(expr) : mstate}$
$\mathsf{init}(e) = (\mathsf{uninit}, \langle\rangle\ \langle\rangle, e)$ if $\mathsf{bverify}(e)$

$\boxed{\mathsf{bverify}(expr) : bool}$
$\mathsf{bverify}(e) = (\mathsf{bsimulate_e}(e, \langle\rangle\ \langle\rangle) \neq \mathsf{invalid})$

$\boxed{\mathsf{bsimulate_e}(expr, bstack) : bstack}$

$$\begin{aligned}
\mathsf{bsimulate_e}(\mathsf{loc}(n), s^b) &= s^b \\
&\quad \text{where} \\
&\quad\quad n < \mathsf{stack\text{-}size}(s^b) \\
&\quad\quad \mathsf{stack\text{-}ref}(s^b, n) = \mathsf{read} \\
\mathsf{bsimulate_e}(\mathsf{lam}(n_{arity}, \overrightarrow{n_{free}}, e), s^b) &= s^b \\
&\quad \text{where} \\
&\quad\quad \overrightarrow{n_{free} < \mathsf{stack\text{-}size}(s^b)} \\
&\quad\quad \overrightarrow{\mathsf{stack\text{-}ref}(s^b, n_{free}) = \mathsf{read}} \\
&\quad\quad \mathsf{bsimulate_e}(e, \langle\rangle\ \langle \overrightarrow{\mathsf{read}}^{|\overrightarrow{n_{free}}|+n_{arity}} \rangle) \neq \mathsf{invalid} \\
\mathsf{bsimulate_e}(\mathsf{app}(e_{fun}, \overrightarrow{e_{args}}), s^b) &= \mathsf{bsimulate^*_e}(e_{fun}\ \overrightarrow{e_{args}}, s^b)
\end{aligned}$$

$\boxed{\mathsf{bsimulate^*_e}(\overrightarrow{expr}, bstack) : bstack}$

$$\begin{aligned}
\mathsf{bsimulate^*_e}(\epsilon, s^b) &= s^b \\
\mathsf{bsimulate^*_e}(e_0\ \overrightarrow{e}, s^b) &= \mathsf{bsimulate^*_e}(\overrightarrow{e}, \mathsf{stack\text{-}pop}(\mathsf{bsimulate_e}(e_0, \mathsf{stack\text{-}push}(s^b))))
\end{aligned}$$

Figure 4.5: Bytecode verification

To express this claim in Modular ACL2, we must first formalize the languages of bytecode expressions and machine states, the behavior of the bytecode verifier, and the process of execution. We specify these features in four interfaces: (1) ifc-bytecode contains the grammar for bytecode expressions and abstract bytecode stacks, (2) ifc-machine describes the language of machine states, (3) ifc-verify presents a theory of bytecode verification, and (4) ifc-execute formalizes machine state execution. A fifth interface, ifc-safety, states the safety property in terms of the functions in the other four.

```
(interface ifc-safety
  (include ifc-machine ifc-verify ifc-initialize ifc-execute)
  (con safety
    (implies
      (and (natp n) (bytecode-expr-p bc) (verify-bytecode-program bc))
      (machine-state-p (machine-execute (machine-initialize bc) n)))))
```

The contract safety states that executing a verified bytecode expression for a given number of steps always results in a machine state (rather than some value outside the grammar of machine states). This statement relies on our transparent definition of the function machine-execute in ifc-execute:

```
(sig app (fun args))
(sig app-p (x))
(sig app.fun (x))
(sig app.args (x))

(con app/boolean
  (booleanp (app-p x))
  :rule-classes :type-prescription)
(con app/predicate
  (app-p (app fun args)))
(con app/constructor
  (implies (app-p x)
    (equal (app (app.fun x) (app.args x)) x))
  :rule-classes :elim)
(con app/selectors
  (and (equal (app.fun (app fun args)) fun)
       (equal (app.args (app fun args)) args)))
(con app/acl2-count
  (and (< (acl2-count fun) (acl2-count (app fun args)))
       (< (acl2-count args) (acl2-count (app fun args)))))
```

Figure 4.6: Specification of application expression data structure

```
(fun machine-execute (ms n)
  (cond
    ((zp n) ms)
    ((error-state-p ms) ms)
    ((endp (registers.control ms)) ms)
    (t (machine-execute (machine-step ms) (- n 1)))))
```

Execution proceeds until n steps have been performed, the machine reaches an error state, or the machine runs out of instructions.

Because the bytecode and machine state grammars share constructors such as loc, lam, and app, these data structures are specified in a separate interface ifc-representation that is included by both ifc-bytecode and ifc-machine. Each constructor is given an associated predicate and accessors, and each data structure is described by a set of contracts.

Figure 4.6 shows the signatures and contracts used for the app data structure. The contracts app/boolean and app/predicate describe the type of the predicate app-p and that it recognizes all app structures. The app/constructor contract permits ACL2 to split any value satisfying app-p into its **fun** and args fields. Both accessors' behavior is specified by app/selectors, and the fact that the value in a field is always smaller than its containing structure is given by app/acl2-count. Note that the type of app's fields are not specified here. The only relationship between data structures specified in ifc-representation is that each structure type is distinct from the others.

The contracts in each interface describe the associated signatures abstractly when practical, leaving out irrelevant implementation details, data representations, and the behavior of functions for unintended inputs. For example, here is the contract in ifc-verify specifying the behavior of bytecode verification for loc expressions:

```
(con verify-bytecode-expr/loc
  (equal (verify-bytecode-expr (loc n) (valid fs))
         (if (and (stack-index-p n fs)
                  (bytecode-readable-p (stack-lookup n fs)))
           (valid fs)
           (invalid))))
```

Given a valid stack, simulation of a loc expression preserves the stack when n refers to an allocated, initialized stack location, and renders it invalid otherwise. This contract provides an algebraic specification of verify-bytecode-expr. The function's inputs are described using the constructors loc and valid; the implementation of dispatch within verify-bytecode-expr and its result for other inputs are not given.

For recursive datatypes and algorithms, we add a transparent function definition to the associated interface that follows the same recursion scheme. This definition permits clients to perform induction using the same scheme. We express mutual recursion, such as that of verify and verify$*$, with a single recursive function containing all of the recursion patterns and choosing among them based on a flag parameter. For instance, the recursion scheme for bytecode expressions and expression lists is shown in figure 4.7.

Our proof proceeds in a top-down fashion; we start with a module importing all of the interfaces specifying the Racket virtual machine and exporting ifc-safety. The natural first step in our proof is induction over the number of steps executed. A naïve attempt, however, results in an inductive hypothesis that is too weak: it is not sufficient to know that machine-state-p holds at one step in order to conclude that it holds at the next. Instead, we need a stronger property that expresses the invariants checked by verify-bytecode-program in terms of machine states.

For this purpose, we specify *machine state verification* in a new interface ifc-simulate. Machine state verification checks properties similar to bytecode verification, such as that every execution step refers to valid stack indices and preserves a well-formed stack.

Abstract machine states consist of an abstract value location and an abstract stack, or may be invalid. An abstract stack is a sequence of abstract stack frames, each of which may be a sequence of abstract value locations or unknown. Abstract value locations may be read-only, write-only, or neither. We once again generalize stack operations to abstract stacks; stack-pop may remove unknown frames, but all other operations are undefined for abstract stacks containing unknown. Figure 4.8 gives the grammar of abstract machine

```
(fun bytecode-recursion (x kind)
  (declare (xargs :measure (bytecode-measure x kind)))
  (cond
    ((equal kind 'bytecode-expr)
     (cond
       ((loc-p x) (loc (loc.s-addr x)))
       ((lam-p x)
        (lam (lam.args x)
             (lam.fvars x)
             (bytecode-recursion (lam.body x) 'bytecode-expr)))
       ((app-p x)
        (app (bytecode-recursion (app.fun x) 'bytecode-expr)
             (bytecode-recursion (app.args x) 'bytecode-expr-list)))
       (t x)))
    ((equal kind 'bytecode-expr-list)
     (cond
       ((null x) nil)
       ((consp x)
        (cons (bytecode-recursion (car x) 'bytecode-expr)
              (bytecode-recursion (cdr x) 'bytecode-expr-list)))
       (t x)))
    (t x)))
```

Figure 4.7: Recursion scheme for expressions and expression lists

$$
\begin{array}{rcll}
m^a \in & astate & = (aloc, astack) \mid \mathsf{invalid} & \text{abstract machine state} \\
s^a \in & astack & = \overrightarrow{aframe} & \text{abstract stack} \\
f^a \in & aframe & = \langle \overrightarrow{aloc} \rangle \mid \mathsf{unknown} & \text{abstract stack frame} \\
\ell^a \in & aloc & = \mathsf{read} \mid \mathsf{write} \mid \mathsf{neither} & \text{abstract value location}
\end{array}
$$

Figure 4.8: Grammar for abstract machine states

states, figure 4.9 presents the convertion from machine states to abstract machine states, and figure 4.10 describes the process of machine state verification.

Given the definition of abstract machine verification, the safety property reduces to two lemmas: passing a verifiable bytecode expression to machine-initialize yields a verifiable machine state, and executing a verifiable machine state yields yet another verifiable machine state. We formalize these two in the interfaces ifc-bytecode-safety and ifc-machine-safety (see figures 4.11 and 4.12), and complete our main theorem by importing these interfaces and supplying appropriate hints to ACL2 in the proof of safety.

The proof of each lemma takes place in its own module. The proof of bytecode-safety proceeds by induction on the verification of the initial machine state. This induction scheme must be developed explicitly for the proof as a new function, as it blends the induction schemes of both bytecode and machine verification; that is, it must account for the "abstract value" field of machine state induction, but it can ignore instructions and intermediate stack

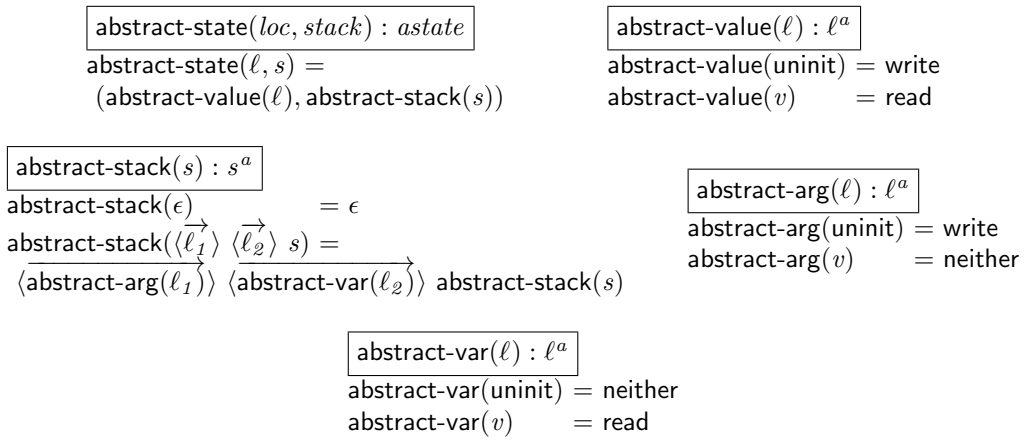

Figure 4.9: Conversion for abstract machine states

states, as an initial machine state cannot contain these. The proof of machine-safety proceeds by induction on the steps in a finite execution trace, and by cases on the possible machine transitions.

At the time of this writing, we have complete and verified implementations of ifc-representation, ifc-bytecode, ifc-machine, ifc-bytecode-safety, and ifc-safety. We have a partial proof of ifc-machine-safety, and have not yet implemented ifc-execute, ifc-verify, or ifc-simulate.

4.3 Experience and Conclusions

In constructing our model and partial safety proof for the Racket virtual machine, we derive several benefits from the use of Modular ACL2 over plain ACL2; in addition, we can identify new directions of improvement for the design and implementation of Modular ACL2.

At the level of component specifications, interfaces allow theories in Modular ACL2 to be "under-specified", leaving out irrelevant aspects of implementations that have nothing to do with the proof at hand. Concretely, this allows ifc-execute to specify transitions only for "good" machine states, while the implementation is free to handle other inputs arbitrarily. This contrasts with the implementation of Klein [2009], which is obligated to ensure that bad inputs invariably either map to bad inputs or get "stuck" so that random testing would yield a false negative if the verifier were overpermissive. Since our proof is in terms of the interface, which guarantees only the behavior of "good" machine states, a successful safety proof thereby assures that verified bytecode yields only "good" machine states. This greatly simplifies our task, as we need neither reason about the behavior of

$$\boxed{\mathsf{mverify}(mstate) : bool}$$

$$\mathsf{mverify}(\ell, s, c) = (\mathsf{msimulate}_{\mathsf{i}}^{*}(\mathsf{abstract\text{-}state}(\ell, s), c) \neq \mathsf{invalid})$$

$$\boxed{\mathsf{msimulate}_{\mathsf{e}}^{*}(astate, \overrightarrow{expr}) : astate}$$

$$\begin{aligned}
\mathsf{msimulate}_{\mathsf{e}}^{*}(m^a, \epsilon) &= m^a \\
\mathsf{msimulate}_{\mathsf{e}}^{*}(m^a, e_0\ \overrightarrow{e}) &= \\
&\mathsf{msimulate}_{\mathsf{e}}^{*}(\mathsf{msimulate}_{\mathsf{e}}(m^a, e_0), \overrightarrow{e})
\end{aligned}$$

$$\boxed{\mathsf{msimulate}_{\mathsf{i}}^{*}(astate, \overrightarrow{instr}) : astate}$$

$$\begin{aligned}
\mathsf{msimulate}_{\mathsf{i}}^{*}(m^a, \epsilon) &= m^a \\
\mathsf{msimulate}_{\mathsf{i}}^{*}(m^a, i_0\ \overrightarrow{i}) &= \\
&\mathsf{msimulate}_{\mathsf{i}}^{*}(\mathsf{msimulate}_{\mathsf{i}}(m^a, i_0), \overrightarrow{i})
\end{aligned}$$

$$\boxed{\mathsf{msimulate}_{\mathsf{e}}(astate, expr) : astate}$$

$$\begin{aligned}
\mathsf{msimulate}_{\mathsf{e}}((\ell_1^a, s_1^a), e) = (&\mathsf{read}, \mathsf{stack\text{-}pop}(s_2^a)) \\
&\text{where} \\
&(\ell_2^a, s_2^a) = \mathsf{msimulate}_{\mathsf{i}}((\ell_1^a, \mathsf{stack\text{-}push}(s_1^a)), e)
\end{aligned}$$

$$\boxed{\mathsf{msimulate}_{\mathsf{i}}(astate, instr) : astate}$$

$$\begin{aligned}
&\mathsf{msimulate}_{\mathsf{i}}((\ell^a, s^a), \mathsf{loc}(n)) = (\mathsf{read}, s^a) \\
&\quad \text{where} \\
&\quad\quad n < \mathsf{stack\text{-}size}(s^a) \\
&\quad\quad \mathsf{stack\text{-}ref}(s^a, n) = \mathsf{read} \\
&\mathsf{msimulate}_{\mathsf{i}}((\ell^a, s^a), \mathsf{lam}(n_{arity}, \overrightarrow{n_{free}}, e)) = (\mathsf{read}, s^a) \\
&\quad \text{where} \\
&\quad\quad \overrightarrow{n_{free} < \mathsf{stack\text{-}size}(s^a)} \\
&\quad\quad \overrightarrow{\mathsf{stack\text{-}ref}(s^a, n_{free}) = \mathsf{read}} \\
&\quad\quad f^a = \langle \overrightarrow{\mathsf{read}}^{|\overrightarrow{n_{free}}| + n_{arity}} \rangle \\
&\quad\quad \mathsf{msimulate}_{\mathsf{i}}((\mathsf{read}, \langle\rangle\ f^a, e)) \neq \mathsf{invalid} \\
&\mathsf{msimulate}_{\mathsf{i}}(m^a, \mathsf{app}(e_{fun}, \overrightarrow{e_{arg}})) = \mathsf{msimulate}_{\mathsf{e}}^{*}(m^a, e_{fun}\ \overrightarrow{e_{arg}}) \\
&\mathsf{msimulate}_{\mathsf{i}}((\ell^a, s^a), \mathsf{push}) = (\ell^a, \mathsf{stack\text{-}push}(s^a)) \\
&\mathsf{msimulate}_{\mathsf{i}}((\ell^a, s^a), \mathsf{pop}) = (\ell^a, \mathsf{stack\text{-}pop}(s^a)) \\
&\mathsf{msimulate}_{\mathsf{i}}((\mathsf{read}, s^a), \mathsf{set}(n)) = (\mathsf{read}, \mathsf{stack\text{-}set}(s^a, n, \mathsf{read})) \\
&\quad \text{where} \\
&\quad\quad n < \mathsf{stack\text{-}size}(s^a) \\
&\quad\quad \mathsf{stack\text{-}ref}(s^a, n) = \mathsf{write} \\
&\mathsf{msimulate}_{\mathsf{i}}((\mathsf{read}, s^a), \mathsf{swap}(n)) = (\mathsf{read}, s^a) \\
&\quad \text{where} \\
&\quad\quad n < \mathsf{stack\text{-}size}(s^a) \\
&\quad\quad \mathsf{stack\text{-}ref}(s^a, n) = \mathsf{neither} \\
&\mathsf{msimulate}_{\mathsf{i}}((\mathsf{read}, s_1^a), \mathsf{call}(n)) = (\mathsf{read}, \langle\rangle\ \mathsf{unknown}\ s_2^a) \\
&\quad \text{where} \\
&\quad\quad s_1^a = \langle \overrightarrow{\mathsf{read}}^n\ \overrightarrow{\ell^a} \rangle\ f^a\ s_2^a \\
&\mathsf{msimulate}_{\mathsf{i}}(m^a, i) = \mathsf{invalid}\ \text{otherwise}
\end{aligned}$$

Figure 4.10: Machine state verification

```
(interface ifc-bytecode-safety
  (include ifc-initialize ifc-verify ifc-simulate)
  (con bytecode-safety
    (implies (and (bytecode-expr-p bc) (verify-bytecode-program bc))
      (and (machine-state-p (machine-initialize bc))
           (verify-machine-state (machine-initialize bc))))))
```

Figure 4.11: Bytecode initialization lemma

```
(interface ifc-machine-safety
  (include ifc-simulate ifc-execute)
  (con machine-safety
    (implies (and (machine-state-p ms) (verify-machine-state ms))
      (and (machine-state-p (machine-execute ms n))
           (verify-machine-state (machine-execute ms n))))))
```

Figure 4.12: Machine state execution lemma

execution of "bad" states, nor detect them in our implementation of execution and produce "bad" results.

Our choice to represent data structures abstractly, focusing on constructors and accessors and ignoring their representations in terms of symbols and lists, yields much more maintainable proofs. The intermediate lemmas generated by ACL2 remain in terms of our virtual machine's data model. For instance, claims about the stack register of a non-invalid abstract machine refer to (result.stack (valid.contents am)). Using a concrete, cons-based implementation yields instead a reference to (cddadr (car am)). This form of expression is not only harder to read, but harder to reason about, as ACL2 cannot apply rewrite rules based on result.stack or valid.contents to claims based solely on car and cdr.

Unfortunately, our abstract data representation imposes a high burden of manual specification. Each data structure requires a sequence of five contracts, similar to those in figure 4.6, which increase in complexity with the number of fields specified. Furthermore, the size of the contracts needed to state disjointness of all structure types increases quadratically in the number of separate data structures. This aspect of development would be significantly improved by direct support for data structure theories in Modular ACL2, so a user need only specify a datatype and all of the necessary contracts would be automatically generated.

Another drawback of Modular ACL2's interfaces occurs at a low level. The bodies of contracts are not compiled by the DrRacket front-end in the current implementation. Instead, they are only compiled by ACL2 at verification time. This can lead to significantly delayed error reports, and even masked errors if an include dependency is omitted, but the

two interfaces happen to be used together most of the time.

Developing proofs and implementations in terms of imported interfaces allows both greater abstraction and more a straightforward method of developing proofs top-down than ACL2 normally affords. Once a proof of a component is completed, changes to the implementation of lower-level components cannot cause Modular ACL2 to "lose" the previously developed proof. In contrast, adding new definitions in the early parts of a proof, or changing a lemma from assumed to proved, alters the logical "world" used by (non-modular) ACL2 and can disrupt proofs that previously worked at a higher level of abstraction. This guarantee of Modular ACL2 adds stability to development; the only time one component must change to suit another is when an interface must be edited.

Verifying a program module-by-module not only enables top-down development; it allows arbitrary interleaving of proof development among different components. For instance, if developing a top-level component raises questions about a lower-level interface, it is easy to switch over to implementing and verifying the low level component to determine what contracts are valid and can be tractably verified of an implementation. We used this technique to ensure our abstract datatype specifications and the mutual induction schemes for our expression grammar were valid; we implemented and maintained these data structures as their specification evolved.

Sadly, the user interface for Modular ACL2 presents a barrier to interleaved editing of different modules and interfaces. Specifically, while verifying a module, Drackula locks all previous definitions—including its imported and exported interfaces—from editing. Any time the output of ACL2 suggests a change to be made to an interface, the current verification progress must be abandoned, the interface edited, and the theorem prover started over from scratch. Dracula and Modular ACL2 would benefit from a user interface that does less to discourage back-and-forth editing during proof development.

Within individual modules, Eastlund and Felleisen [2009a] claim that Modular ACL2 can obviate the need for most or all proof annotations such as ACL2's "hints" to optimize the performance of the theorem proving engine. Our proof of the bytecode safety lemma required hints for approximately two-thirds of all of its intermediate lemmas. These hints are necessary to make the proof complete successfully; they are not used strictly to improve speed.

At intermediate stages of proof development, some of our efforts to provide an abstract specification of imported components may have been counterproductive. Our attempts to provide an underspecified theory of verification and evaluation, ignoring their behavior on "bad" inputs, requires each contract to have extra hypotheses constraining each free variable to the minimum set of values required for correctness. Unfortunately, using these contracts

during verification requires ACL2 to discharge all of these hypotheses. This can significantly slow down the verification process; it can also cause ACL2 to pass up opportunities to use a contract because its hypotheses are not "obviously" true. In a few cases, we decided to make our contracts less abstract and more flexible in the interests of pragmatic theorem proving.

Reasoning in terms of data constructors rather than accessors—for instance, claiming that x and y are integers in (cons x y) rather than (car p) and (cdr p) in p when (consp p) holds—makes writing straightforward, abstract contracts easy, but it often makes completing proofs in ACL2 harder. Function definitions and induction schemes that inspect values in ACL2 are based on predicates and accessors, not constructors. While ACL2 does use so-called "elimination rules" to expand constrained variables (e.g., p when (consp p) holds) into constructor expressions (e.g., (cons x y)), these rules are applied late in the process of rewriting a claim. ACL2 therefore misses many opportunities to apply lemmas written in terms of constructors. In order to make use of these lemmas, the user must either write many explicit proof hints or else derive predicate-based corollaries from the constructor-based contracts.

Chapter 5

Refining Modules and Macros

Modular ACL2 provides reusable specifications but does not support a mechanism to abstract over them. In this chapter we describe Refined ACL2, which adapts Modular ACL2 in two ways to address this problem: with macros, which abstract over syntax—including that of specifications—and with refinements, which extend specifications to describe a particular implementation in detail.

As in Modular ACL2, Refined ACL2 has an executable semantics and a verification semantics. The executable semantics takes the form of a step relation that removes component constructs from the program one at a time, eventually yielding a plain ACL2 program. The verification semantics translates a Refined ACL2 program into an ACL2 verification obligation. We also formalize a static semantics for Refined ACL2 and ACL2, which ensures syntacic and logical well-formedness.

5.1 Example

We present an example program in the macro-extensible surface syntax of Refined ACL2 in figure 5.1. The first two definitions present two types: TYPE and LISTOF.

The type named TYPE specifies component instances implementing predicates. The predicate must be a unary function named is? and must be accompanied by a proof of the theorem is?/boolean?, which states that is? produces a boolean result.

The type LISTOF specifies instances implementing homogenous lists. Each implementation of LISTOF must include a nested instance named Elem, a unary function filter, and a theorem filter/member; the type also supplies two macros named add and make. The nested instance Elem must implement TYPE in order to provide a type predicate for list elements. The filter function must be unary and must satisfy the stated theorem filter/member meaning any elements of its result must satisfy the predicate is? from Elem. The macro add conditionally adds an element to the front of a list, returning the given list unless the

```
(description TYPE
  (stub (is? x))
  (theorem (is?/boolean? x)
    (boolean? (is? x))))

(description LISTOF
  (component Elem : TYPE)
  (stub (filter xs))
  (theorem (filter/member x xs)
    (implies (member x (filter xs))
      (Elem.is? x)))
  (macro add
    [(_ e1 e2)
     (let {[x e1] [xs e2]}
       (if (Elem.is? x) (cons x xs) xs))])
  (macro make
    [(_) '()]
    [(_ e1 e2 ...) (add e1 (make e2 ...))]))

(generic (Listof E : TYPE) : LISTOF where {[Elem = E]}
  (function (filter xs)
    (cond
      [(empty? xs) xs]
      [(cons? xs) (add (first xs) (filter (rest xs)))])))

(component Integer : TYPE where {[is? = integer?]})

(instance Listof-Integer (Listof Integer))

(Listof-Integer.make 1 "two")
```

Figure 5.1: Example Refined ACL2 program.

new element satisfies Elem.is?. Full lists can be constructed with the macro make, which expands into nested uses of add. Macros are always made available for use by clients of the description. For convenience, macros made available to providers of a description as well.

The third definition provides Listof, a generic component accepting an implementation of TYPE. The generic's result is sealed at the type LISTOF refined by a **where** clause, exposing that the nested instance Elem is implemented by the generic's argument E. The body of Listof implements filter; Elem is already implemented via description refinement, and the theorem filter/member required by LISTOF is implicitly defined with no proof annotations. The body of filter uses the macro add; in this context, Elem.is? in the macro's output refers to the Elem refined to be equal to E.

We implement TYPE in the fourth definition. The instance Integer refines TYPE to specify that its member is? is implemented by integer?.

Finally, we instantiate Listof as Listof-Integer using Integer as its argument and construct

$$
\begin{array}{l}
p \coloneqq \overrightarrow{d} \\
d \coloneqq i = t \\
t \coloneqq a \\
\quad \mid \mathsf{fun}\, i(\overrightarrow{i})\; e\; e\; \overrightarrow{h} \\
\quad \mid \mathsf{thm}\, (\overrightarrow{i})\; e\; \overrightarrow{r}\; \overrightarrow{h} \\
\quad \mid \mathsf{inst}\{\overrightarrow{\ell \triangleright d}\} \\
\quad \mid \mathsf{gen}\, (i : T)\; t \\
\quad \mid t(a) \\
\quad \mid \mathsf{seal}\, t : T \\
\quad \mid \mathsf{type}\, T
\end{array}
\qquad
\begin{array}{l}
P \coloneqq \overrightarrow{D} \\
D \coloneqq i : T \\
T \coloneqq a \\
\quad \mid \mathsf{stub}\, (n) \\
\quad \mid \mathsf{fun}\, i(\overrightarrow{i})\; e\; e \\
\quad \mid \mathsf{thm}\, (\overrightarrow{i})\; e\; \overrightarrow{r} \\
\quad \mid \mathsf{inst}\{\overrightarrow{\ell \triangleright D}\} \\
\quad \mid \mathsf{gen}\, (i : T)\; T \\
\quad \mid T \mathsf{where}\, \overrightarrow{\ell} = a \\
\quad \mid \mathsf{type}\, T \\
\quad \mid \mathsf{ref}\, a \\
\quad \mid \mathsf{value}
\end{array}
\qquad
\begin{array}{l}
a \coloneqq i \\
\quad \mid a \centerdot \ell \\
e \coloneqq i \\
\quad \mid n \\
\quad \mid \mathsf{if}\; e\; e\; e \\
\quad \mid \mathsf{let}\; i = e\; \mathsf{in}\; e \\
\quad \mid a(\overrightarrow{e}) \\
r \coloneqq e\; \mathsf{equals}\; e\; \mathsf{when}\; e \\
\quad \mid e\; \mathsf{recurs\,as}\; e \\
h \coloneqq \mathsf{lemma}\; a(\overrightarrow{e}) \\
\quad \mid \mathsf{induct}\; e
\end{array}
$$

Figure 5.2: Grammar of Refined ACL2.

a list using Listof-Integer.make. In this context, the reference to add in the expansion of make refers to Listof-Integer.add, and the reference to Elem.is? in add refers to Listof-Integer.Elem.is?.

5.2 Core Language

Macros extend Refined ACL2; before verification and execution, we must expand a source program into a macro-free core language. Expansion also effectively alpha-renames the source programs; in a fully-expanded program, there is no shadowing. Figure 5.2 shows the grammar of this core. A program p is a sequence of definitions; each definition d assigns a name to a term.

Terms, t, denote ACL2 definitions, components that combine them, and types that describe them. A term may be an address referring to an existing definition. Terms also include functions and theorems à la ACL2. A function definition consists of an identifier used internally for recursive references, a sequence of identifiers naming formal arguments, expressions for its body and measure, and a sequence of hints to aid the theorem prover. Theorems have universally quantified identifiers, a body expression, a sequence of rule classes, and a sequence of hints. Components may be instances, which contain a sequence of labelled definitions, as well as generics, which have a formal argument with a specified type as well as a body term. Application terms provide a term representing a generic with an actual argument specified by an address. A component may also be sealed at a given type, possibly hiding details of its implementation. Finally, types themselves can be encapsulated as terms.

Types, T, describe the contribution of terms to an ACL2 logical theory. Types include

references to defined types. A type may describe a function of a given arity for which nothing else is known, denoted stub(n). A type may also describe a complete function or theorem, denoted the same as their terms except that proof hints are not included. Component instances have corresponding types, mapping labels to declarations. The type of a generic has a formal argument and input and output types. A type can be *refined*, specifying an implementation by address for the base type or one of its members. Refined members may be deeply nested; a member's location is specified by a sequence of labels in order from outermost to innermost. An empty sequence of labels signifies a refinement to the entire base type. Encapsulated types serve as their own types. The type of references to a previously defined function or theorem are given nominally. Finally, value bindings are described simply as value.

Just as types describe terms, environments, P, and declarations, D, describe programs and definitions. Their forms are analogous.

An address, a, can be either an identifier or a reference to a labelled member of another address. Expressions in function and theorem bodies can be variable references, constants, conditionals, lexical bindings, and function applications. A rule class can be an equational rewrite or a recursion-scheme association. Finally, proof hints may invoke a lemma or may specify induction scheme. Lemmas are specified as an application of a function or theorem name to explicit arguments. Induction schemes are given as expressions.

We macro-expand the example from figure 5.1 to the core Refined ACL2 language in figures 5.3 and 5.4.

In the definition of TYPE, we assign labels to each declaration using the same name as the surface identifier, and alpha-rename the identifiers to is?1 and is?/boolean?1. In is?/boolean?1 we also rename the formal argument and add the default rewrite rule class based on the theorem's body.

Labels and fresh identifiers are added to LISTOF similarly. The theorem filter/member1 gets a default rewrite rule class as well, and references to the dotted identifier Elem.is? are converted to references to the member labelled is? in Elem1. Note that make and add are not present in the elaborated description; macros are all expanded at their application sites during the elaboration process.

The generic Listof and its argument E have unique identifiers and do not need to be renamed. We assign the names Elem2, filter2, and filter/member2 to the contents of Listof and map the labels of LISTOF to these identifiers. Elaboration adds an alias from Elem2 to E based on the refinement of LISTOF in the range of Listof. The function filter2 gets the default measure of 0 and an internal name of filter3 for recursion. The conditional in its body becomes a chain of if expressions terminating with a call to void at the end. The

```
TYPE =
  type
    inst
    { is? ▷ is?1 : stub (1);
      is?/boolean? ▷ is?/boolean?1 :
        thm (x1)
          boolean?(is?1(x1))
          boolean?(is?1(x1)) equals #true when #true }

LISTOF =
  type
    inst
    { Elem ▷ Elem1 : TYPE;
      filter ▷ filter1 : stub (1);
      filter/member ▷ filter/member1 :
        thm (x2, xs1)
          implies(member(x2, filter1(xs1)), Elem1 . is?(x2))
          Elem1 . is?(x2) equals #true when #true }

Listof =
  gen (E : TYPE)
    seal
      inst
      { Elem ▷ Elem2 = E;
        filter ▷ filter2 =
          fun filter3(xs2)
            if cons?(xs2)
              let x3 = first(xs2)
              in let xs3 = filter3(rest(xs2))
                 in if Elem2 . is?(x3) cons(x3, xs3) xs3
              void()
            0
            ε;
        filter/member ▷ filter/member2 =
          thm (x4, xs4)
            implies(member(x4, filter2(xs4)), Elem2 . is?(x4))
            Elem2 . is?(x4) equals #true when #true
            ε }
    : LISTOF where Elem = E
```

Figure 5.3: Expanded version of example Refined ACL2 program, part 1.

void function is defined as a stub; Refined ACL2 cannot reason about what specific value cond produces if all clauses fail. The application of add in the original definition of filter is expanded; the reference to Elem.is? in the original becomes Elem2 . is? inside Listof. The definition of filter/member2 is filled in with no proof hints, and a body and rule classes based on the description LISTOF.

In the definition of Integer, we assign the names is?2 and is?/boolean?2 to the internal

```
Integer =
  seal
    inst
    { is? ▷ is?2 = integer?;
      is?/boolean? ▷ is?/boolean?2 =
        thm (x5)
          boolean?(is?2(x5))
          boolean?(is?2(x5)) equals #true when #true
          ε }
  : TYPE where is? = integer?

Listof-Integer = Listof(Integer)

let x6 = 1
in let xs5 =
     let x7 = "two"
     in let xs6 = null()
       in if Listof-Integer . Elem . is?(x7) cons(x7, xs6) xs6
  in if Listof-Integer . Elem . is?(x6) cons(x6, xs5) xs5
```

Figure 5.4: Expanded version of example Refined ACL2 program, part 2.

definitions for the members labelled is? and is?/boolean?. We elaborate the internal definitions to add an automatic alias for the refined declaration of is? and an automatic, hint-free definition for is?/boolean?2.

Finally, we expand the reference to Listof-Integer.make, which generates references to Listof-Integer.add based on the add in the original macro definition. We expand these in turn, alpha-renaming all introduced identifiers, and producing the final expression.

5.3 Static Semantics

The static semantics for Refined ACL2 ensures that each part of a program is sufficiently well-formed for verification: identifier references must be unambiguous, arguments to generics must implement the expected type, and so forth. We also disallow shadowing in all contexts. Macro-expanded programs never shadow bindings, so input programs always satisfy this constraint; preserving this constraint aids us in defining our verification and execution semantics. The type of each definition describes a contribution to the program's overall logical theory; the type-checking rules build up a theory one type at a time.

The typing rules for Refined ACL2 programs and definitions appear in figure 5.5. The judgment $P_1 \vdash_{\mathrm{p}} p :: P_2$ means that in the context of environment P_1, the program p contributes P_2 to the current logical theory. The empty program contributes nothing to a theory. In a non-empty program, the tail of the program is type-checked in a context

$$\boxed{P \vdash_{\mathrm{p}} p :: P}$$

$$\frac{}{P \vdash_{\mathrm{p}} \varepsilon :: \varepsilon} \qquad \frac{P_1 \vdash_{\mathrm{p}} d :: D \qquad P_1; D \vdash_{\mathrm{p}} p :: P_2}{P_1 \vdash_{\mathrm{p}} d; p :: D; P_2}$$

$$\boxed{P \vdash_{\mathrm{d}} d :: D}$$

$$\frac{\begin{array}{c} i_0, i_1, \overrightarrow{i_2} \text{ distinct} \qquad i_0, i_1, \overrightarrow{i_2} \notin \mathrm{dom}(P_1) \\ P_2 = P_1; i_1 : \mathsf{stub}\,(|\overrightarrow{i_2}|); \overrightarrow{i_2 : \mathsf{value}} \\ P_2 \vdash_{\mathrm{e}} e_1 \qquad P_2 \vdash_{\mathrm{e}} e_2 \qquad \overrightarrow{P_2 \vdash_{\mathrm{h}} h} \end{array}}{P_1 \vdash_{\mathrm{d}} i_0 = \mathsf{fun}\, i_1(\overrightarrow{i_2})\, e_1\, e_2\, \overrightarrow{h} :: i_0 : \mathsf{fun}\, i_1(\overrightarrow{i_2})\, e_1\, e_2}$$

$$\frac{\begin{array}{c} i_0, \overrightarrow{i_1} \text{ distinct} \qquad i_0, \overrightarrow{i_1} \notin \mathrm{dom}(P_1) \\ P_2 = P_1; \overrightarrow{i_1 : \mathsf{value}} \\ P_2 \vdash_{\mathrm{e}} e \qquad \overrightarrow{P_2 \vdash_{\mathrm{r}} r} \qquad \overrightarrow{P_2 \vdash_{\mathrm{h}} h} \end{array}}{P_1 \vdash_{\mathrm{d}} i_0 = \mathsf{thm}\,(\overrightarrow{i_1})\, e\, \overrightarrow{r}\, \overrightarrow{h} :: i_0 : \mathsf{thm}\,(\overrightarrow{i_1})\, e\, \overrightarrow{r}}$$

$$\frac{i \notin \mathrm{dom}(P) \qquad P \vdash_{\mathrm{t}} t :: T}{P \vdash_{\mathrm{d}} i = t :: i : T}$$

Figure 5.5: Typing rules for Refined ACL2 programs and definitions.

extended by the declaration of the first definition's type. The whole program's contribution appends that of the definition and the program's tail.

A definition is mapped to a declaration, given an environment as context. This judgment is denoted $P \vdash_{\mathrm{d}} d :: D$. Functions and theorems in Refined ACL2 are nominal entities; because they are identified by their names, they must appear directly on the right-hand side of a definition. In both cases, all of their introduced identifiers must be unique and unbound. Each of their subcomponents—expressions, rule classes, and hints—must be well-formed in an environment where their formals and, in the case of functions, their local names are bound. Both functions and theorems are assigned a type that is equivalent to their term except that the hints are dropped. For other terms, the term is typechecked independently and the full definition is assigned the term's type.

Figure 5.6 shows the rules that assign types to Refined ACL2 terms and addresses. A term is assigned a type in a given environment according to the judgment $P \vdash_{\mathrm{t}} t :: T$.

When a term is an address, its type is looked up in the environment, then refined to state its name. This refinement during lookup allows types to be stored in the environment without referring to their own names, an important invariant for other judgments.

Instance terms are assigned a type based on the types of the definitions in their body, so long as their labels are unique. The mapping of labels to declarations assumes that the

$$\boxed{P \vdash_{\mathrm{t}} t :: T}$$

$$\frac{P \vdash_{\mathrm{a}} a :: T}{P \vdash_{\mathrm{t}} a :: T \,\mathsf{where}\, \varepsilon = a} \qquad \frac{\overrightarrow{\ell}\ \text{distinct} \qquad P \vdash_{\mathrm{p}} \overrightarrow{d} :: \overrightarrow{D}}{P \vdash_{\mathrm{t}} \mathsf{inst}\{\overrightarrow{\ell \triangleright d}\} :: \mathsf{inst}\{\overrightarrow{\ell \triangleright D}\}}$$

$$\frac{\begin{array}{c} P \vdash_{\mathrm{T}} T_1 \qquad P \vdash_{\mathrm{T}} T_1 \leqslant \mathsf{inst}\{\varepsilon\} \\ i \notin \mathrm{dom}(P) \qquad P; i : T_1 \vdash_{\mathrm{t}} t :: T_2 \\ P; i : T_1 \vdash_{\mathrm{T}} T_2 \leqslant \mathsf{inst}\{\varepsilon\} \end{array}}{P \vdash_{\mathrm{t}} \mathsf{gen}\,(i : T_1)\, t :: \mathsf{gen}\,(i : T_1)\, T_2} \qquad \frac{\begin{array}{c} P \vdash_{\mathrm{t}} a :: T_1 \qquad P \vdash_{\mathrm{t}} t :: T_2 \\ P \vdash_{\mathrm{T}} T_2 \hookrightarrow \mathsf{gen}\,(i : T_3)\, T_4 \qquad P \vdash_{\mathrm{T}} T_1 \leqslant T_3 \end{array}}{P \vdash_{\mathrm{t}} t(a) :: [a/i]\, T_4}$$

$$\frac{\begin{array}{c} P \vdash_{\mathrm{T}} T_1 \qquad P \vdash_{\mathrm{T}} T_1 \leqslant \mathsf{inst}\{\varepsilon\} \\ P \vdash_{\mathrm{t}} t :: T_2 \qquad P \vdash_{\mathrm{T}} T_2 \leqslant T_1 \end{array}}{P \vdash_{\mathrm{t}} \mathsf{seal}\, t : T_1 :: T_1} \qquad \frac{P \vdash_{\mathrm{T}} T}{P \vdash_{\mathrm{t}} \mathsf{type}\, T :: \mathsf{type}\, T}$$

$$\boxed{P \vdash_{\mathrm{a}} a :: T}$$

$$\frac{i : T \in P}{P \vdash_{\mathrm{a}} i :: T} \qquad \frac{P \vdash_{\mathrm{a}} a :: T_1 \qquad P \vdash_{\mathrm{T}} T_1 \hookrightarrow \mathsf{inst}\{\overrightarrow{\ell_1 \triangleright i_1 : T_1}; \ell_0 \triangleright i_0 : T_0; \overrightarrow{\ell_2 \triangleright i_2 : T_2}\}}{P \vdash_{\mathrm{a}} a \,.\, \ell_0 :: [\overrightarrow{a.\ell_1/i_1}]\, T_0}$$

Figure 5.6: Typing rules for Refined ACL2 types and addresses.

type judgment for definitions produces a sequence of declarations in the same order.

For a generic term to be well-typed, several conditions must hold. The generic's formal argument must be unbound and its input type must be well-formed. In an environment where the formal argument has the input type, the generic's body has to be well-typed. We further restrict the input and the body to have instance types by requiring an upper bound of $\mathsf{inst}\{\varepsilon\}$. This restriction keeps the input and output of generics from having nominal types of the form $\mathsf{ref}\ a$; we rely on this restriction in our soundness proof.

To typecheck the application of a generic, we first determine the type of the generic and its argument. The generic's type must in fact reduce to an appropriate generic type, and the argument's type must be a subtype of the generic's domain. The type of the application term is the generic's output type with the argument address substituted for the generic's formal argument.

Sealing a term at a given type requires the term to be well-typed and the sealed type to be well-formed. The argument term's type must be a subtype of the sealed type; the entire sealing term is assigned the sealed type.

For any well-formed type T, the term $\mathsf{type}\ T$ serves as its own type.

The judgment $P \vdash_{\mathrm{a}} a :: T$ looks up the type assigned to a given address by the current environment. For an identifier, this requires merely looking up the appropriate declaration. Finding the type of a member reference entails recursively finding the type of the base ad-

$$\boxed{P \vdash_{\mathrm{P}} P}$$

$$\frac{}{P \vdash_{\mathrm{P}} \varepsilon} \qquad \frac{P_1 \vdash_{\mathrm{D}} D \qquad P_1; D \vdash_{\mathrm{P}} P_2}{P_1 \vdash_{\mathrm{P}} D; P_2}$$

$$\boxed{P \vdash_{\mathrm{D}} D}$$

$$\frac{i \notin \mathrm{dom}(P) \qquad P \vdash_{\mathrm{T}} T}{P \vdash_{\mathrm{D}} i : T} \qquad \frac{i \notin \mathrm{dom}(P)}{P \vdash_{\mathrm{D}} i : \mathsf{stub}\,(n)}$$

$$\frac{\begin{array}{c} i_0, i_1, \overrightarrow{i_2} \text{ distinct} \qquad i_0, i_1, \overrightarrow{i_2} \notin \mathrm{dom}(P_1) \\ P_2 = P_1; i_1 : \mathsf{stub}\,(|\overrightarrow{i_2}|); \overrightarrow{i_2 : \mathsf{value}} \\ P_2 \vdash_{\mathrm{e}} e_1 \qquad P_2 \vdash_{\mathrm{e}} e_1 \end{array}}{P_1 \vdash_{\mathrm{D}} i_0 : \mathsf{fun}\, i_1 (\overrightarrow{i_2})\, e_1\, e_2} \qquad \frac{\begin{array}{c} i_0, \overrightarrow{i_1} \text{ distinct} \qquad i_0, \overrightarrow{i_1} \notin \mathrm{dom}(P_1) \\ P_2 = P_1; \overrightarrow{i_1 : \mathsf{value}} \\ P_2 \vdash_{\mathrm{e}} e \qquad \overrightarrow{P_2 \vdash_{\mathrm{r}} r} \end{array}}{P_1 \vdash_{\mathrm{D}} i_0 : \mathsf{thm}\,(\overrightarrow{i_1})\, e\, \overrightarrow{r}}$$

$$\boxed{P \vdash_{\mathrm{T}} T}$$

$$\frac{P \vdash_{\mathrm{a}} a :: T_1 \qquad P \vdash_{\mathrm{T}} T_1 \hookrightarrow \mathsf{type}\ T_2}{P \vdash_{\mathrm{T}} a} \qquad \frac{}{P \vdash_{\mathrm{T}} \mathsf{value}} \qquad \frac{\overrightarrow{\ell} \text{ distinct} \qquad P \vdash_{\mathrm{P}} \overrightarrow{D}}{P \vdash_{\mathrm{T}} \mathsf{inst}\{\overrightarrow{\ell \triangleright D}\}}$$

$$\frac{\begin{array}{c} i \notin \mathrm{dom}(P) \qquad P \vdash_{\mathrm{T}} T_1 \qquad P \vdash_{\mathrm{T}} T_1 \leqslant \mathsf{inst}\{\varepsilon\} \\ P; i : T_1 \vdash_{\mathrm{T}} T_2 \qquad P; i : T_1 \vdash_{\mathrm{T}} T_2 \leqslant \mathsf{inst}\{\varepsilon\} \end{array}}{P \vdash_{\mathrm{T}} \mathsf{gen}\,(i : T_1)\, T_2}$$

$$\frac{P \vdash_{\mathrm{t}} a :: T_1 \qquad P \vdash_{\mathrm{T}} T_2 \qquad P \vdash_{\mathrm{T}} T_1 \leqslant T_2 \mathbin{@} \overrightarrow{\ell}}{P \vdash_{\mathrm{T}} T_1 \,\mathsf{where}\, \overrightarrow{\ell} = a} \qquad \frac{P \vdash_{\mathrm{T}} T}{P \vdash_{\mathrm{T}} \mathsf{type}\ T}$$

$$\frac{P \vdash_{\mathrm{a}} a :: T \qquad P \vdash_{\mathrm{T}} T \leqslant \mathsf{stub}\,(n)}{P \vdash_{\mathrm{T}} \mathsf{ref}\ a} \qquad \frac{P \vdash_{\mathrm{a}} a :: T \qquad P \vdash_{\mathrm{T}} T \leqslant \mathsf{thm}\,(\overrightarrow{i})\, e\, \overrightarrow{r}}{P \vdash_{\mathrm{T}} \mathsf{ref}\ a}$$

Figure 5.7: Typing rules for Refined ACL2 environments, declarations, and types.

dress, then reducing it to an instance type. This instance type must contain the appropriate label. Its type is returned, with all references to local names from the instance replaced with appropriate external member references.

The well-formedness of environments, declarations, and types largely follows the same pattern as typechecking programs, definitions, and terms, as shown in figure 5.7.

An environment P_2 is determined to be well-formed in an enclosing environment P_1 by the judgment $P_1 \vdash_{\mathrm{P}} P_2$. An empty environment is always well-formed; declarations are typechecked individually and accumulate in the environment.

The judgment $P \vdash_{\mathrm{D}} D$ states that a declaration is well-formed in a given environment. As with the corresponding definitions, function and theorem declarations require their introduced names to be unique and unbound and their contents to be well-formed in an

environment including their local bindings. Stub declarations are well-formed so long as their name is unbound. Other types in declarations must be well-formed on their own for the declaration as a whole to be well-formed.

Well-formed types are determined by the judgment $P \vdash_{\mathrm{T}} T$. Addresses as types must refer to components whose type reduces to $\mathsf{type}\ T$.

The type value is well-formed.

An instance type is well-formed when its labels are distinct and its declarations are well-formed as an environment.

A generic is well-formed if its formal argument is unbound, its input type is well-formed, its output type is well-formed when its argument is bound, and both the input and output type are subtypes of $\mathsf{inst}\{\varepsilon\}$.

For a refined type to be well-formed, the specified address must be well-typed and the base type must be well-formed. Furthermore, the address's type must be a subtype of the specified member's type, as denoted by the judgment $P \vdash_{\mathrm{T}} T_1 \leqslant T_2 \mathbin{@} \overrightarrow{\ell}$.

The type of an encapsulated type is well-formed if the contained type is well-formed.

A nominal reference type's address must have a function or theorem type.

The type reduction relation $P \vdash_{\mathrm{T}} T_1 \hookrightarrow T_2$ serves to reduce T_1 to T_2 by looking up references and resolving refinements according to P.

When the type of an address reduces to $\mathsf{type}\ T$, that address itself reduces to T.

Instance types can be reduced by reducing their constituent declarations as an environment and reconstructing the instance type with the result.

A generic type reduces by recursively reducing its input and output types.

A stub, function, or theorem type refined to a specific name with an empty sequence of labels reduces to a reference type.

Refined instance types reduce by "pushing" the refinement inward to the types of members. When the refinement has at least one label, the refinement is transferred to the member corresponding to the first label; that member is refined with the remaining labels and the original address. A refinement with no labels is distributed across all members, with the refined address for each updated to dereference the corresponding label. Furthermore, all references to local names in the instance type are updated to refer to the appropriate member of the refined address.

Reduction simply drops refinements from generic types, reference types, and the types of types themselves.

The type of a type-component reduces according to the reduction of the contained type.

Reduction can "chase" aliases in reference types back to the original reference.

Finally, type reduction is reflexive and transitive.

$$\boxed{P \vdash_{\mathrm{T}} T \hookrightarrow T}$$

$$\frac{P \vdash_{\mathrm{a}} a :: T_1 \qquad P \vdash_{\mathrm{T}} T_1 \hookrightarrow \mathsf{type}\ T_2}{P \vdash_{\mathrm{T}} a \hookrightarrow T_2} \qquad \frac{P \vdash_{\mathrm{P}} \overrightarrow{D_1} \hookrightarrow \overrightarrow{D_2}}{P \vdash_{\mathrm{T}} \mathsf{inst}\{\overrightarrow{\ell \triangleright D_1}\} \hookrightarrow \mathsf{inst}\{\overrightarrow{\ell \triangleright D_2}\}}$$

$$\frac{P \vdash_{\mathrm{T}} T_1 \hookrightarrow T_3 \qquad P; i : T_3 \vdash_{\mathrm{T}} T_2 \hookrightarrow T_4}{P \vdash_{\mathrm{T}} \mathsf{gen}\,(i : T_1)\ T_2 \hookrightarrow \mathsf{gen}\,(i : T_3)\ T_4} \qquad \frac{}{P \vdash_{\mathrm{T}} \mathsf{stub}\,(n)\,\mathsf{where}\,\varepsilon = a \hookrightarrow \mathsf{ref}\ a}$$

$$\frac{}{P \vdash_{\mathrm{T}} \mathsf{fun}\ i_1(\overrightarrow{i_2})\ e_1\ e_2\,\mathsf{where}\,\varepsilon = a \hookrightarrow \mathsf{ref}\ a} \qquad \frac{}{P \vdash_{\mathrm{T}} \mathsf{thm}\,(\overrightarrow{i_1})\ e\ \overrightarrow{r}\,\mathsf{where}\,\varepsilon = a \hookrightarrow \mathsf{ref}\ a}$$

$$\frac{}{\begin{array}{c}P \vdash_{\mathrm{T}} \mathsf{inst}\{\overrightarrow{\ell_1 \triangleright D_1}; \ell_3 \triangleright i : T; \overrightarrow{\ell_2 \triangleright D_2}\}\,\mathsf{where}\,\ell_3, \overrightarrow{\ell_4} = a \\ \hookrightarrow \mathsf{inst}\{\overrightarrow{\ell_1 \triangleright D_1}; \ell_3 \triangleright i : T\,\mathsf{where}\,\overrightarrow{\ell_4} = a; \overrightarrow{\ell_2 \triangleright D_2}\}\end{array}}$$

$$\frac{}{P \vdash_{\mathrm{T}} \mathsf{inst}\{\overrightarrow{\ell \triangleright i : T}\}\,\mathsf{where}\,\varepsilon = a \hookrightarrow \mathsf{inst}\{\overrightarrow{\ell \triangleright i : [\overrightarrow{a.\ell}/i]\,T\,\mathsf{where}\,\varepsilon = a\,.\,\ell}\}}$$

$$\frac{}{P \vdash_{\mathrm{T}} \mathsf{gen}\,(i : T_1)\ T_2\,\mathsf{where}\,\varepsilon = a \hookrightarrow \mathsf{gen}\,(i : T_1)\ T_2} \qquad \frac{}{P \vdash_{\mathrm{T}} \mathsf{type}\ T\,\mathsf{where}\,\varepsilon = a \hookrightarrow \mathsf{type}\ T}$$

$$\frac{}{P \vdash_{\mathrm{T}} \mathsf{ref}\ a_1\,\mathsf{where}\,\varepsilon = a_2 \hookrightarrow \mathsf{ref}\ a_1} \qquad \frac{P \vdash_{\mathrm{T}} T_1 \hookrightarrow T_2}{P \vdash_{\mathrm{T}} \mathsf{type}\ T_1 \hookrightarrow \mathsf{type}\ T_2} \qquad \frac{P \vdash_{\mathrm{a}} a_1 :: \mathsf{ref}\ a_2}{P \vdash_{\mathrm{T}} \mathsf{ref}\ a_1 \hookrightarrow \mathsf{ref}\ a_2}$$

$$\frac{}{P \vdash_{\mathrm{T}} T \hookrightarrow T} \qquad \frac{P \vdash_{\mathrm{T}} T_1 \hookrightarrow T_2 \qquad P \vdash_{\mathrm{T}} T_2 \hookrightarrow T_3}{P \vdash_{\mathrm{T}} T_1 \hookrightarrow T_3}$$

$$\boxed{P \vdash_{\mathrm{P}} P \hookrightarrow P}$$

$$\frac{}{P \vdash_{\mathrm{P}} \varepsilon \hookrightarrow \varepsilon} \qquad \frac{P_0 \vdash_{\mathrm{T}} T_1 \hookrightarrow T_2 \qquad P_0; i : T_2 \vdash_{\mathrm{P}} P_1 \hookrightarrow P_2}{P_0 \vdash_{\mathrm{P}} i : T_1; P_1 \hookrightarrow i : T_2; P_2}$$

Figure 5.8: Type reduction for Refined ACL2.

Environment reduction is denoted $P_1 \vdash_{\mathrm{P}} P_2 \hookrightarrow P_3$. This signifies that in environment P_1, P_2 reduces to P_3. The empty environment reduces to itself; non-empty environments are reduced one declaration at a time, binding each name to its reduced type in turn.

The relation $P \vdash_{\mathrm{T}} T_1 \leqslant T_2 \mathbin{@} \overrightarrow{\ell}$ states that in environment P, T_1 is a subtype of the potentially deeply-nested member of T_2 found by successively dereferencing the labels in $\overrightarrow{\ell}$, which are ordered from outermost to innermost. When the sequence of labels are empty, this reduces directly to subtyping.

When there is at least one label, T_2 must reduce to an instance type containing the first label to dereference. Member-subtyping proceeds by introducing the local type bindings from the instance to the environment, then comparing T_1 to the appropriate member type using the remaining labels. We assign fresh names to the new bindings and rename the

$$\boxed{P \vdash_{\mathrm{T}} T \leqslant T \mathbin{@} \overrightarrow{\ell}}$$

$$\frac{P \vdash_{\mathrm{T}} T_1 \leqslant T_2}{P \vdash_{\mathrm{T}} T_1 \leqslant T_2 \mathbin{@} \varepsilon} \qquad \frac{\begin{array}{c} P \vdash_{\mathrm{T}} T_2 \hookrightarrow \mathsf{inst}\{\overrightarrow{\ell_3 \triangleright i_3 : T_3}; \ell_1 \triangleright i_1 : T_0; \overrightarrow{\ell_4 \triangleright D}\} \\ \overrightarrow{i_0} \text{ fresh} \qquad P; i_0 : [\overrightarrow{i_0/i_3}]\overrightarrow{T_3} \vdash_{\mathrm{T}} T_1 \leqslant [\overrightarrow{i_0/i_3}]T_0 \mathbin{@} \overrightarrow{\ell_2} \end{array}}{P \vdash_{\mathrm{T}} T_1 \leqslant T_2 \mathbin{@} \ell_1, \overrightarrow{\ell_2}}$$

$$\boxed{P \vdash_{\mathrm{T}} T \leqslant T}$$

$$\frac{}{P \vdash_{\mathrm{T}} T \leqslant T} \qquad \frac{P \vdash_{\mathrm{T}} T_1 \leqslant T_2 \qquad P \vdash_{\mathrm{T}} T_2 \leqslant T_3}{P \vdash_{\mathrm{T}} T_1 \leqslant T_3}$$

$$\frac{P \vdash_{\mathrm{T}} T_1 \hookrightarrow T_3 \qquad P \vdash_{\mathrm{T}} T_2 \hookrightarrow T_4 \qquad P \vdash_{\mathrm{T}} T_3 \leqslant T_4}{P \vdash_{\mathrm{T}} T_1 \leqslant T_2}$$

$$\frac{}{P \vdash_{\mathrm{T}} \mathsf{fun}\, i_1(\overrightarrow{i_2})\, e_1\, e_2 \leqslant \mathsf{stub}\,(|\overrightarrow{i_2}|)} \qquad \frac{\begin{array}{c} |\overrightarrow{i_2}| = |\overrightarrow{i_4}| \qquad P_2 = P_1; i_1 : \mathsf{stub}\,(|\overrightarrow{i_2}|); \overrightarrow{i_2 : \mathsf{value}} \\ P_2 \vdash_{\mathrm{e}} e_1 \equiv [i_1/i_3, \overrightarrow{i_2/i_4}]e_2 \qquad P_2 \vdash_{\mathrm{e}} e_3 \equiv [i_1/i_3, \overrightarrow{i_2/i_4}]e_4 \end{array}}{P_1 \vdash_{\mathrm{T}} \mathsf{fun}\, i_1(\overrightarrow{i_2})\, e_1\, e_2 \leqslant \mathsf{fun}\, i_3(\overrightarrow{i_4})\, e_3\, e_4}$$

$$\frac{\begin{array}{c} |\overrightarrow{i_1}| = |\overrightarrow{i_2}| \qquad P_2 = P_1; \overrightarrow{i_1 : \mathsf{value}} \\ P_2 \vdash_{\mathrm{e}} e_1 \equiv [\overrightarrow{i_1/i_2}]e_2 \qquad P_2 \vdash_{\mathrm{r}} r_1 \equiv [\overrightarrow{i_1/i_2}]r_2 \end{array}}{P_1 \vdash_{\mathrm{T}} \mathsf{thm}\,(\overrightarrow{i_1})\, e_1\, \overrightarrow{r_1} \leqslant \mathsf{thm}\,(\overrightarrow{i_2})\, e_2\, \overrightarrow{r_2}}$$

$$\frac{\begin{array}{c} \{\overrightarrow{\ell_1}\} \supseteq \{\overrightarrow{\ell_2}\} \qquad \overrightarrow{\ell_2 \triangleright i_3 : T_3} \subseteq \overrightarrow{\ell_1 \triangleright i_1 : T_1} \\ \overrightarrow{i_4} \text{ fresh} \qquad P; \overrightarrow{i_4 : [\overrightarrow{i_4/i_1}]T_1} \vdash_{\mathrm{T}} [\overrightarrow{i_4/i_1}]T_3 \leqslant [\overrightarrow{i_4/i_1}][\overrightarrow{i_3/i_2}]T_2 \end{array}}{P \vdash_{\mathrm{T}} \mathsf{inst}\{\overrightarrow{\ell_1 \triangleright i_1 : T_1}\} \leqslant \mathsf{inst}\{\overrightarrow{\ell_2 \triangleright i_2 : T_2}\}}$$

$$\frac{P \vdash_{\mathrm{T}} T_3 \leqslant T_1 \qquad i_3 \text{ fresh} \qquad P; i_3 : T_3 \vdash_{\mathrm{T}} [i_3/i_1]T_2 \leqslant [i_3/i_2]T_4}{P \vdash_{\mathrm{T}} \mathsf{gen}\,(i_1 : T_1)\, T_2 \leqslant \mathsf{gen}\,(i_2 : T_3)\, T_4} \qquad \frac{P \vdash_{\mathrm{a}} a :: T}{P \vdash_{\mathrm{T}} \mathsf{ref}\, a \leqslant T}$$

Figure 5.9: Subtyping for Refined ACL2.

member type accordingly in order to avoid reusing any names also bound in T_1.

We determine whether T_1 is a subtype of T_2 in environment P with the judgment $P \vdash_{\mathrm{T}} T_1 \leqslant T_2$. Subtyping is reflexive and transitive, and two types are subtypes if they have reduced forms that are subtypes.

Any function type is a subtype of the stub with the same arity.

One function type is a subtype of another if they are α-equivalent, as determined by renaming the local name and formal arguments of one to match the other and comparing their body and measure expressions.

Similar to function types, one theorem type is a subtype of another if they are α-equivalent. Once again, we rename one type's formal arguments and compare the corre-

$$\boxed{P \vdash_{\mathrm{h}} h}$$

$$\frac{P \vdash_{\mathrm{t}} a :: T \qquad \overrightarrow{P \vdash_{\mathrm{e}} e} \qquad P \vdash_{\mathrm{T}} T \leqslant \mathsf{stub}\,(|\overrightarrow{e}|)}{P \vdash_{\mathrm{h}} \mathsf{lemma}\ a(\overrightarrow{e})} \qquad \frac{P \vdash_{\mathrm{t}} a :: T \qquad \overrightarrow{P \vdash_{\mathrm{e}} e_1} \qquad P \vdash_{\mathrm{T}} T \leqslant \mathsf{thm}\,(\overrightarrow{i})\ e_2\ \overrightarrow{r} \qquad |\overrightarrow{i}| = |\overrightarrow{e_1}|}{P \vdash_{\mathrm{h}} \mathsf{lemma}\ a(\overrightarrow{e_1})} \qquad \frac{P \vdash_{\mathrm{e}} e}{P \vdash_{\mathrm{h}} \mathsf{induct}\ e}$$

$$\boxed{P \vdash_{\mathrm{r}} r}$$

$$\frac{P \vdash_{\mathrm{e}} e_1 \qquad P \vdash_{\mathrm{e}} e_2 \qquad P \vdash_{\mathrm{e}} e_3}{P \vdash_{\mathrm{r}} e_1\ \mathsf{equals}\ e_2\ \mathsf{when}\ e_3} \qquad \frac{P \vdash_{\mathrm{e}} e_1 \qquad P \vdash_{\mathrm{e}} e_2}{P \vdash_{\mathrm{r}} e_1\ \mathsf{recurs\ as}\ e_2}$$

$$\boxed{P \vdash_{\mathrm{e}} e}$$

$$\frac{P \vdash_{\mathrm{a}} i :: \mathsf{value}}{P \vdash_{\mathrm{e}} i} \qquad \frac{}{P \vdash_{\mathrm{e}} n} \qquad \frac{P \vdash_{\mathrm{e}} e_1 \qquad i \notin \mathrm{dom}(P) \qquad P; i : \mathsf{value} \vdash_{\mathrm{e}} e_2}{P \vdash_{\mathrm{e}} \mathsf{let}\ i = e_1\ \mathsf{in}\ e_2}$$

$$\frac{P \vdash_{\mathrm{e}} e_1 \qquad P \vdash_{\mathrm{e}} e_2 \qquad P \vdash_{\mathrm{e}} e_3}{P \vdash_{\mathrm{e}} \mathsf{if}\ e_1\ e_2\ e_3} \qquad \frac{P \vdash_{\mathrm{a}} a :: T \qquad P \vdash_{\mathrm{T}} T \leqslant \mathsf{stub}\,(|\overrightarrow{e}|) \qquad \overrightarrow{P \vdash_{\mathrm{e}} e}}{P \vdash_{\mathrm{e}} a(\overrightarrow{e})}$$

Figure 5.10: Typing rules for Refined ACL2 expressions.

sponding expression and rule classes.

Instance subtyping is complex. The subtype must contain all the labels from the supertype. The comparison proceeds by binding all of the declarations from the subtype in the environment under fresh names. We then compare the member types from the supertype to the corresponding members of the subtype, renamed to use the aforementioned fresh identifiers.

Generic subtyping compares input types contravariantly and output types covariantly; when comparing output types, a fresh name is bound to the formal argument of the supertype, and both output types are renamed to use the fresh name in place of their corresponding formal argument.

A reference type is always a subtype of the type recorded for its address.

Typing rules for expressions, hints, and rule classes appear in figure 5.10. The judgment $P \vdash_{\mathrm{h}} h$ determines hint well-formedness. In a lemma hint, the argument expressions must be well-formed and the lemma address must have a function or theorem type of the appropriate arity. The expression in an induction hint must be well-formed.

Rule classes are well-formed when their subexpressions are well-formed, according to the judgment $P \vdash_{\mathrm{r}} r$.

Expression well-formedness requires variable references to be bound as values, lexical binders to be unbound in the enclosing environment, and applied functions to be bound at

$$\boxed{P \vdash_{\mathrm{r}} r \equiv r}$$

$$\frac{P \vdash_{\mathrm{e}} e_1 \equiv e_4 \qquad P \vdash_{\mathrm{e}} e_2 \equiv e_5 \qquad P \vdash_{\mathrm{e}} e_3 \equiv e_6}{P \vdash_{\mathrm{r}} e_1 \ \mathsf{equals}\ e_2 \ \mathsf{when}\ e_3 \equiv e_4 \ \mathsf{equals}\ e_5 \ \mathsf{when}\ e_6} \qquad \frac{P \vdash_{\mathrm{e}} e_1 \equiv e_3 \qquad P \vdash_{\mathrm{e}} e_2 \equiv e_4}{P \vdash_{\mathrm{r}} e_1 \ \mathsf{recurs\ as}\ e_2 \equiv e_3 \ \mathsf{recurs\ as}\ e_4}$$

$$\boxed{P \vdash_{\mathrm{e}} e \equiv e}$$

$$\frac{}{P \vdash_{\mathrm{e}} i \equiv i} \qquad \frac{}{P \vdash_{\mathrm{e}} n \equiv n}$$

$$\frac{P \vdash_{\mathrm{e}} e_1 \equiv e_3 \qquad i_3 \text{ fresh} \qquad P; i_3 : \mathsf{value} \vdash_{\mathrm{e}} [i_3/i_1]e_2 \equiv [i_3/i_2]e_4}{P \vdash_{\mathrm{e}} \mathsf{let}\ i_1 = e_1 \ \mathsf{in}\ e_2 \equiv \mathsf{let}\ i_2 = e_3 \ \mathsf{in}\ e_4}$$

$$\frac{P \vdash_{\mathrm{e}} e_1 \equiv e_4 \qquad P \vdash_{\mathrm{e}} e_2 \equiv e_5 \qquad P \vdash_{\mathrm{e}} e_3 \equiv e_6}{P \vdash_{\mathrm{e}} \mathsf{if}\ e_1\ e_2\ e_3 \equiv \mathsf{if}\ e_4\ e_5\ e_6}$$

$$\frac{P \vdash_{\mathrm{T}} \mathsf{ref}\ a_1 \hookrightarrow \mathsf{ref}\ a_3 \qquad P \vdash_{\mathrm{T}} \mathsf{ref}\ a_2 \hookrightarrow \mathsf{ref}\ a_3 \qquad \overrightarrow{P \vdash_{\mathrm{e}} e_1 \equiv e_2}}{P \vdash_{\mathrm{e}} a_1(\overrightarrow{e_1}) \equiv a_2(\overrightarrow{e_2})}$$

Figure 5.11: Equivalence rules for Refined ACL2 expressions.

function types with the appropriate arity. All this is determined by the relation $P \vdash_{\mathrm{e}} e$.

Figure 5.11 describes the judgments for equivalence and well-formedness of hints, rule classes, and expressions. The judgment $P \vdash_{\mathrm{r}} r_1 \equiv r_2$ determines rule class equivalence by comparing respective subexpressions.

Two expressions are equivalent according to the judgment $P \vdash_{\mathrm{e}} e_1 \equiv e_2$ if they are α-equivalent; function references are compared using reference type reduction to track aliases.

The typing rules for ACL2 programs are given in figure 5.12. The judgments $\overline{P}_1 \vdash_{\overline{\mathrm{p}}} \overline{p} :: \overline{P}_2$ and $\overline{P}_1 \vdash_{\overline{\mathrm{d}}} \overline{d} :: \overline{P}_2$ determine the type of programs, function declarations, and theorem declarations analogously to their Refined ACL2 counterparts.

ACL2 additionally allows stub definitions, which have the corresponding stub type so long as their name is unbound.

A set of hidden definitions must be well-typed, but does not contribute to the environment of the rest of the program.

Assumed definitions, by contrast, contribute directly to the surrounding environment.

Figure 5.13 defines well-formedness of ACL2 hints, rule classes, and expressions, which proceed analogously to those in Refined ACL2. There are two key differences. First, as there are no components, identifiers suffice in place of addresses and all types can be directly looked up in the environment. Second, we define subtyping only for environments and declarations, not for types. As a result, function and lemma applications directly test for stub, function, and theorem types in the environment.

$$\boxed{\overline{P} \vdash_{\overline{\mathrm{p}}} \overline{p} :: \overline{P}}$$

$$\frac{}{\overline{P} \vdash_{\overline{\mathrm{p}}} \varepsilon :: \varepsilon} \qquad \frac{\overline{P}_1 \vdash_{\overline{\mathrm{d}}} \overline{d} :: \overline{P}_2 \qquad \overline{P}_1; \overline{P}_2 \vdash_{\overline{\mathrm{p}}} \overline{p} :: \overline{P}_3}{\overline{P}_1 \vdash_{\overline{\mathrm{p}}} \overline{d}; \overline{p} :: \overline{P}_2; \overline{P}_3}$$

$$\boxed{\overline{P} \vdash_{\overline{\mathrm{d}}} \overline{d} :: \overline{P}}$$

$$\frac{i \notin \mathrm{dom}(\overline{P})}{\overline{P} \vdash_{\overline{\mathrm{d}}} i = \mathsf{stub}\,(n) :: i : \mathsf{stub}\,(n)}$$

$$\frac{\begin{array}{c} i_0, i_1, \overrightarrow{i_2} \text{ distinct} \qquad i_0, i_1, \overrightarrow{i_2} \notin \mathrm{dom}(\overline{P}_1) \\ \overline{P}_2 = \overline{P}_1; i_1 : \mathsf{stub}\,(|\overrightarrow{i_2}|); \overrightarrow{i_2 : \mathsf{value}} \qquad \overline{P}_2 \vdash_{\overline{\mathrm{e}}} \overline{e}_1 \qquad \overline{P}_2 \vdash_{\overline{\mathrm{e}}} \overline{e}_2 \qquad \overrightarrow{\overline{P}_2 \vdash_{\overline{\mathrm{h}}} \overline{h}} \end{array}}{\overline{P}_1 \vdash_{\overline{\mathrm{d}}} i_0 = \mathsf{fun}\, i_1(\overrightarrow{i_2})\, \overline{e}_1\, \overline{e}_2\, \overrightarrow{\overline{h}} :: i_0 : \mathsf{fun}\, i_1(\overrightarrow{i_2})\, \overline{e}_1\, \overline{e}_2}$$

$$\frac{\begin{array}{c} i_0, \overrightarrow{i_1} \text{ distinct} \qquad i_0, \overrightarrow{i_1} \notin \mathrm{dom}(\overline{P}_1) \\ \overline{P}_2 = \overline{P}_1; \overrightarrow{i_1 : \mathsf{value}} \qquad \overline{P}_2 \vdash_{\overline{\mathrm{e}}} \overline{e} \qquad \overrightarrow{\overline{P}_2 \vdash_{\overline{\mathrm{r}}} \overline{r}} \qquad \overrightarrow{\overline{P}_2 \vdash_{\overline{\mathrm{h}}} \overline{h}} \end{array}}{\overline{P}_1 \vdash_{\overline{\mathrm{d}}} i_0 = \mathsf{thm}\,(\overrightarrow{i_1})\, \overline{e}\, \overrightarrow{\overline{r}}\, \overrightarrow{\overline{h}} :: i_0 : \mathsf{fun}\, \overrightarrow{i_1}\,(\overline{e})\, \overrightarrow{\overline{r}}} \qquad \frac{\overline{P}_1 \vdash_{\overline{\mathrm{p}}} p :: \overline{P}_2}{\overline{P}_1 \vdash_{\overline{\mathrm{d}}} \mathsf{hidden}\,\{p\} :: \varepsilon}$$

$$\frac{\overline{P}_1 \vdash_{\overline{\mathrm{p}}} p :: \overline{P}_2}{\overline{P}_1 \vdash_{\overline{\mathrm{d}}} \mathsf{assume}\,\{p\} :: \overline{P}_2}$$

Figure 5.12: Typing rules for ACL2 programs and definitions.

Subtyping in ACL2 proceeds in terms of environments and declarations, and is used in determining valid functional instantiations; see figure 5.14. The judgment $\overline{P}_1 \vdash_{\overline{\mathrm{P}}} \overline{P}_2 \leqslant \overrightarrow{i}\ @\,\overline{P}_3$ states that in environment $\overline{P}_1$, $\overline{P}_2$ is a valid functional instantiation for $\overline{P}_3$, using $\overrightarrow{i}$ as the instantiations for the names in $\overline{P}_3$. The comparison proceeds by determining whether the union of $\overline{P}_1$ and $\overline{P}_2$ entails each declaration in $\overline{P}_3$, renamed to the instantiated identifiers.

Declaration entailment is written $\overline{P} \models_{\overline{\mathrm{D}}} \overline{D}$. Any declaration is entailed by an environment if the environment maps the same name to an alpha-equivalent type. As we disallow shadowing, alpha-equivalence is defined straightforwardly. A stub declaration is also entailed if the environment contains a corresponding function with the same arity.

Figure 5.15 gives a few relevant rules for proof entailment in ACL2; we only formalize those aspects of ACL2's logic that are relevant to the soundness of Refined ACL2. The judgment $\overline{P} \models_{\overline{\mathrm{p}}} \overline{p}$ means that environment $\overline{P}$ represents a logical theory that entails the proof obligations of $\overline{p}$.

Proving a set of definitions proceeds one definition at a time, adding the type of each to the environment in turn. Furthermore, if one environment entails a proof, any functional

$$\boxed{\overline{P} \vdash_{\overline{\mathrm{h}}} \overline{h}}$$

$$\frac{i : \mathsf{stub}\,(|\overrightarrow{\overline{e}}|) \in \overline{P} \qquad \overrightarrow{\overline{P} \vdash_{\overline{\mathrm{e}}} \overline{e}}}{\overline{P} \vdash_{\overline{\mathrm{h}}} \mathsf{lemma}\; i(\overrightarrow{\overline{e}})} \qquad \frac{i_0 : \mathsf{fun}\; i_1(\overrightarrow{i_2})\; \overline{e}_1\; \overline{e}_2 \in \overline{P} \qquad |\overrightarrow{i_2}| = |\overrightarrow{\overline{e}_0}| \qquad \overrightarrow{\overline{P} \vdash_{\overline{\mathrm{e}}} \overline{e}_0}}{\overline{P} \vdash_{\overline{\mathrm{h}}} \mathsf{lemma}\; i_0(\overrightarrow{\overline{e}_0})}$$

$$\frac{i_0 : \mathsf{thm}\,(\overrightarrow{i_1})\; \overline{e}_1\; \overrightarrow{r} \in \overline{P} \qquad |\overrightarrow{i_1}| = |\overrightarrow{\overline{e}_0}| \qquad \overrightarrow{\overline{P} \vdash_{\overline{\mathrm{e}}} \overline{e}_0}}{\overline{P} \vdash_{\overline{\mathrm{h}}} \mathsf{lemma}\; i_0(\overrightarrow{\overline{e}_0})} \qquad \frac{\overline{P} \vdash_{\overline{\mathrm{e}}} \overline{e}}{\overline{P} \vdash_{\overline{\mathrm{h}}} \mathsf{induct}\; \overline{e}}$$

$$\boxed{\overline{P} \vdash_{\overline{\mathrm{r}}} \overline{r}}$$

$$\frac{\overline{P} \vdash_{\overline{\mathrm{e}}} \overline{e}_1 \qquad \overline{P} \vdash_{\overline{\mathrm{e}}} \overline{e}_2 \qquad \overline{P} \vdash_{\overline{\mathrm{e}}} \overline{e}_3}{\overline{P} \vdash_{\overline{\mathrm{r}}} \overline{e}_1\; \mathsf{equals}\; \overline{e}_2\; \mathsf{when}\; \overline{e}_3} \qquad \frac{\overline{P} \vdash_{\overline{\mathrm{e}}} \overline{e}_1 \qquad \overline{P} \vdash_{\overline{\mathrm{e}}} \overline{e}_2}{\overline{P} \vdash_{\overline{\mathrm{r}}} \overline{e}_1\; \mathsf{recurs\ as}\; \overline{e}_2}$$

$$\boxed{\overline{P} \vdash_{\overline{\mathrm{e}}} \overline{e}}$$

$$\frac{i : \mathsf{value} \in \overline{P}}{\overline{P} \vdash_{\overline{\mathrm{e}}} i} \qquad \frac{}{\overline{P} \vdash_{\overline{\mathrm{e}}} n} \qquad \frac{\overline{P} \vdash_{\overline{\mathrm{e}}} \overline{e}_1 \qquad \overline{P} \vdash_{\overline{\mathrm{e}}} \overline{e}_2 \qquad \overline{P} \vdash_{\overline{\mathrm{e}}} \overline{e}_3}{\overline{P} \vdash_{\overline{\mathrm{e}}} \mathsf{if}\; \overline{e}_1\; \overline{e}_2\; \overline{e}_3}$$

$$\frac{\overline{P} \vdash_{\overline{\mathrm{e}}} \overline{e}_1 \qquad i \notin \mathrm{dom}(\overline{P}) \qquad \overline{P}; i : \mathsf{value} \vdash_{\overline{\mathrm{e}}} \overline{e}_2}{\overline{P} \vdash_{\overline{\mathrm{e}}} \mathsf{let}\; i = \overline{e}_1\; \mathsf{in}\; \overline{e}_2} \qquad \frac{i : \mathsf{stub}\,(|\overrightarrow{\overline{e}}|) \in \overline{P} \qquad \overrightarrow{\overline{P} \vdash_{\overline{\mathrm{e}}} \overline{e}}}{\overline{P} \vdash_{\overline{\mathrm{e}}} i(\overrightarrow{\overline{e}})}$$

$$\frac{i_0 : \mathsf{fun}\; i_1(\overrightarrow{i_2})\; \overline{e}_1\; \overline{e}_2 \in \overline{P} \qquad |\overrightarrow{i_2}| = |\overrightarrow{\overline{e}_0}| \qquad \overrightarrow{\overline{P} \vdash_{\overline{\mathrm{e}}} \overline{e}_0}}{\overline{P} \vdash_{\overline{\mathrm{e}}} i_0(\overrightarrow{\overline{e}_0})}$$

Figure 5.13: Typing rules for ACL2 hints, rule classes, and expressions.

instantiation of a portion of that environment entails the same proof renamed according to the instantiation.

The judgment $\overline{P} \models_{\overline{\mathrm{d}}} \overline{d}$ describes when a definition is provable from a given environment. We do not formalize provability for stub, function, and theorem definitions; our design is not dependent on ACL2's rules for atomic definitions. Assumed definitions are always provable; hidden definitions are provable whenever their content is provable.

To demonstrate the typechecking process, we examine the elaborated program from figures 5.3 and 5.4. To typecheck the program, we must assign a type to each definition. We assume an initial environment with built-in functions like integer? and boolean?, and an extended grammar that includes boolean constants #true and #false.

The type component TYPE can be used as its own type so long as it is well-formed, meaning its constituent declarations must be well-formed. Since the instance type's labels and names are all unique and the function and variable references in is?/boolean?1 are all bound, the type is well-formed. The resulting declaration for TYPE follows:

$$\boxed{\overline{P} \vdash_{\overline{\mathrm{P}}} \overline{P} \leqslant \overline{P} \mathbin{@} \overrightarrow{i}}$$

$$\frac{\overline{\overline{P}_1; \overline{P}_2 \models_{\overline{\mathrm{D}}} i_1 : [\overrightarrow{i_1/i_2}]\overrightarrow{T}}}{\overline{P}_1 \vdash_{\overline{\mathrm{P}}} \overline{P}_2 \leqslant \overrightarrow{i_2 : T} \mathbin{@} \overrightarrow{i_1}}$$

$$\boxed{\overline{P} \models_{\overline{\mathrm{D}}} \overline{D}}$$

$$\frac{i : \overline{T}_2 \in \overline{P} \qquad \overline{T}_1 \stackrel{\alpha}{=} \overline{T}_2}{\overline{P} \models_{\overline{\mathrm{D}}} i : \overline{T}_1} \qquad\qquad \frac{i_0 : \mathsf{fun}\, i_1(\overrightarrow{i_2})\, e_1\, e_2 \in \overline{P}}{\overline{P} \models_{\overline{\mathrm{D}}} i_0 : \mathsf{stub}\,(|\overrightarrow{i_2}|)}$$

Figure 5.14: Subtyping rules for ACL2.

$$\boxed{\overline{P} \models_{\overline{\mathrm{p}}} \overline{p}}$$

$$\frac{}{\overline{P} \models_{\overline{\mathrm{p}}} \varepsilon} \qquad \frac{\overline{P}_1 \models_{\overline{\mathrm{d}}} \overline{d} \qquad \overline{P}_1 \vdash_{\overline{\mathrm{d}}} \overline{d} :: \overline{P}_2 \qquad \overline{P}_1; \overline{P}_2 \models_{\overline{\mathrm{p}}} \overline{p}}{\overline{P}_1 \models_{\overline{\mathrm{p}}} \overline{d}; \overline{p}}$$

$$\frac{\overline{P}_1; \overrightarrow{i_1 : T} \models_{\overline{\mathrm{p}}} \overline{p} \qquad \overline{P}_1 \vdash_{\overline{\mathrm{P}}} \overline{P}_2 \leqslant \overrightarrow{i_1 : T} \mathbin{@} \overrightarrow{i_2}}{\overline{P}_1; \overline{P}_2 \models_{\overline{\mathrm{p}}} [\overrightarrow{i_2/i_1}]\overline{p}}$$

$$\boxed{\overline{P} \models_{\overline{\mathrm{d}}} \overline{d}}$$

$$\frac{}{\overline{P} \models_{\overline{\mathrm{p}}} \mathsf{assume}\,\{\overline{p}\}} \qquad \frac{\overline{P} \models_{\overline{\mathrm{p}}} \overline{p}}{\overline{P} \models_{\overline{\mathrm{p}}} \mathsf{hidden}\,\{\overline{p}\}}$$

Figure 5.15: Selected proof rules for ACL2.

```
TYPE :
  type
    inst
    { is? ▷ is?1 : stub (1);
      is?/boolean? ▷ is?/boolean?1 :
        thm (x1)
          boolean?(is?1(x1))
          boolean?(is?1(x1)) equals #true when #true }
```

We address the same considerations for LISTOF. Once again, the instance type's labels and names are unique and unbound. The reference to TYPE is legal, since it has been type-checked; stub declarations are always well-formed; and the function and variable references in filter/member are all valid. The result is:

```
LISTOF :
  type
    inst
    { Elem ▷ Elem1 : TYPE;
      filter ▷ filter1 : stub (1);
      filter/member ▷ filter/member1 :
        thm (x2, xs1)
          implies(member(x2, filter1(xs1)), Elem1 . is?(x2))
          Elem1 . is?(x2) equals #true when #true }
```

In order to typecheck Listof, we must check the body of the generic in an environment containing a declaration for E:

```
E : TYPE
```

The body of Listof is an instance sealed at a specified type. Typechecking begins by assigning a type to the instance term, going through its definitions one by one. The first definition in the instance is an alias for E. When referring to E, we refine its type to identify its nominal elements. Thus Elem2 is declared at the following type:

```
Elem2 : TYPE where ε = E
```

Next, the typechecking process encounters filter2, defined as a function. We ensure that its bound identifiers are unique and that its body and measure are well-formed. The declaration for filter2 assigns it a function type, dropping the empty sequence of hints from the function term:

```
filter2 :
  fun filter3(xs2)
    if cons?(xs2)
      let x3 = first(xs2)
      in let xs3 = filter3(rest(xs2))
        in if Elem2 . is?(x3) cons(x3, xs3) xs3
      void()
    0
```

Typechecking for filter/member2 proceeds similarly; its constituent parts are well-formed and it is assigned a theorem type:

```
filter/member2 :
  thm (x4, xs4)
    implies(member(x4, filter1(xs4)), Elem2 . is?(x4))
    Elem2 . is?(x4) equals #true when #true
```

We arrive at the following type for the instance term in Listof, reducing the type of Elem2 to show the types of its members:

```
inst
{ Elem ▷ Elem2 :
    inst
    { is? ▷ is?1 : ref E . is?;
      is?/boolean? ▷ is?/boolean?1 : ref E . is?/boolean? };
  filter ▷ filter2 :
    fun filter3(xs2)
      if cons?(xs2)
        let x3 = first(xs2)
        in let xs3 = filter3(rest(xs2))
             in if Elem2 . is?(x3) cons(x3, xs3) xs3
        void()
      0;
  filter/member ▷ filter/member2 :
    thm (x4, xs4)
      implies(member(x4, filter1(xs4)), Elem2 . is?(x4))
      Elem2 . is?(x4) equals #true when #true }
```

To finish typechecking the sealing term, we must determine whether its explicit type is well-formed and whether its term has a subtype of the explicit type. The reference to LISTOF is valid as a type; the refinement equating the member Elem to the reference E is valid because the type of E is a subtype of the member Elem of LISTOF. We arrive at the following reduced type at which to seal the body of Listof:

```
inst
{ Elem ▷ Elem1 :
    inst
    { is? ▷ is?1 : ref E . is?;
      is?/boolean? ▷ is?/boolean?1 : ref E . is?/boolean? };
  filter ▷ filter1 : stub (1);
  filter/member ▷ filter/member1 :
    thm (x2, xs1)
      implies(member(x2, filter1(xs1)), Elem1 . is?(x2))
      Elem1 . is?(x2) equals #true when #true }
```

We must compare the two previous types to establish a subtype relationship. To do this, we bind the contents of the instance term's type in an extended environment. Using that environment, we compare the type of each corresponding pair of members. The Elem members both have identical types. The instance's function type for filter is sealed at a stub type of the same arity. The theorem types for filter/member are α-equivalent to each other. The generic Listof can therefore be assigned a generic type:

```
Listof : gen (E : TYPE) LISTOF where Elem = E
```

The sealing term for Integer must be checked in the same way as the body of Listof. We arrive at the following type for the nested instance term.

```
inst
{ is? ▷ is?2 : ref integer?;
  is?/boolean? ▷ is?/boolean?2 :
    thm (x5)
      boolean?(is?2(x5))
      boolean?(is?2(x5)) equals #true when #true }
```

When reduced, the type at which Integer is sealed appears as follows.

```
inst
{ is? ▷ is?1 : ref integer?;
  is?/boolean? ▷ is?/boolean?1 :
    thm (x1)
      boolean?(is?1(x1))
      boolean?(is?1(x1)) equals #true when #true }
```

Both declarations for is? use the same type, and again the theorem types for the final declarations are α-equivalent. We can thus add Integer to the top-level environment.

```
Integer : TYPE where is? = integer?
```

The final definition applies Listof to Integer. The typechecking process ensures that Integer's type is a subtype of Listof's expected input type. We arrive finally at a type for Listof-Integer.

```
Listof-Integer : LISTOF where Elem = Integer
```

5.4 Verification Semantics

Refined ACL2 verification proceeds much like verification in Modular ACL2. We convert each program to an ACL2 program that expresses the necessary proof obligations. Unlike in Modular ACL2, we need produce only one program for verification. We use encapsulated ACL2 definitions to logically separate parts of the program from each other.

We formalize a core subset of ACL2 in order to describe proof obligations for the Refined ACL2 language. For the most part, our ACL2 grammar is the same as Refined ACL2 without components, their types, or member references.

The grammar for ACL2 used in proof obligations appears in figure 5.16. ACL2 programs, $\overline{p}$, are sequences of definitions. A definition, $\overline{d}$, can assign a name to a term as in Refined ACL2. It can also specify a set of hidden, to-be-verified definitions whose names and logical consequences are not available to the rest of the program. Finally, a definition may describe a set of assumed definitions that are not mechanically verified, but instead trusted and added directly to the current logical theory.

$$
\begin{array}{lll}
\overline{p} \coloneqq \overrightarrow{\overline{d}} & \overline{P} \coloneqq \overrightarrow{\overline{D}} & \overline{e} \coloneqq i \\
\overline{d} \coloneqq i = \overline{t} & \overline{D} \coloneqq i : \overline{T} & \quad \mid\ n \\
\quad \mid\ \mathsf{hidden}\,\{\overline{p}\} & \overline{T} \coloneqq \mathsf{stub}\,(n) & \quad \mid\ \mathsf{if}\ \overline{e}\ \overline{e}\ \overline{e} \\
\quad \mid\ \mathsf{assume}\,\{\overline{p}\} & \quad \mid\ \mathsf{fun}\ i(\overrightarrow{i})\ \overline{e}\ \overline{e} & \quad \mid\ \mathsf{let}\ i = \overline{e}\ \mathsf{in}\ \overline{e} \\
\overline{t} \coloneqq \mathsf{stub}\,(n) & \quad \mid\ \mathsf{thm}\,(\overrightarrow{i})\ \overline{e}\ \overrightarrow{\overline{r}} & \quad \mid\ i(\overrightarrow{\overline{e}}) \\
\quad \mid\ \mathsf{fun}\ i(\overrightarrow{i})\ \overline{e}\ \overline{e}\ \overrightarrow{\overline{h}} & & \overline{r} \coloneqq \overline{e}\ \mathsf{equals}\ \overline{e}\ \mathsf{when}\ \overline{e} \\
\quad \mid\ \mathsf{thm}\,(\overrightarrow{i})\ \overline{e}\ \overrightarrow{\overline{r}}\ \overrightarrow{\overline{h}} & & \quad \mid\ \overline{e}\ \mathsf{recurs\ as}\ \overline{e} \\
 & & \overline{h} \coloneqq \mathsf{lemma}\ i(\overrightarrow{\overline{e}}) \\
 & & \quad \mid\ \mathsf{induct}\ \overline{e}
\end{array}
$$

Figure 5.16: Grammar of ACL2.

$$
\begin{array}{l}
\Sigma \coloneqq \overrightarrow{i \mapsto \sigma} \\
\sigma \coloneqq \mathsf{none} \\
\quad \mid\ \mathsf{self} \\
\quad \mid\ \mathsf{alias}\ i \\
\quad \mid\ \mathsf{inst}\,\{\overrightarrow{\ell \triangleright i}\}
\end{array}
$$

Figure 5.17: Verification environment for Refined ACL2.

Terms in ACL2, $\overline{t}$, can be functions with or without bodies, or theorems. Types for ACL2 programs mirror terms, except that there are no special types for hidden and assumed definitions. Expressions, rule classes, and hints in ACL2 are exactly the same as in Refined ACL2.

Figure 5.17 introduces *shapes* and *shape mappings*. A shape describes the ACL2 name or names corresponding to a particular term in Refined ACL2. Shapes can be none for terms that are erased or hidden in the program's verification obligation. The shape self describes function and theorem terms whose name is preserved. A reference to a previously defined term named i is assigned the shape $\mathsf{alias}\ i$. Finally, instance shapes map labels to ACL2 names. A shape mapping maps names to shapes.

Figure 5.18 describes the rules that produce verification obligations for programs, definitions, and terms. The judgment $P, \Sigma_1 \vdash_{\mathrm{p}} p \rightsquigarrow \Sigma_2, \overline{p}$ constructs a verification obligation for the program p in the context of P and Σ_1. This translation generates a new shape mapping, Σ_2 and proof obligations, $\overline{p}$. The empty program produces an empty shape mapping and an empty proof obligation. Definitions are verified one at a time, with their types and shape mappings added to the context of the rest of the program.

Individual definitions in Refined ACL2 can produce multiple ACL2 definitions in their proof obligations. The judgment $P, \Sigma_1 \vdash_{\mathrm{d}} d \rightsquigarrow \Sigma_2, \overline{p}$ reflects this, as it produces a shape mapping and ACL2 program fragment for every Refined ACL2 definition in the context of

$$\boxed{P, \Sigma \vdash_{\mathrm{p}} p \rightsquigarrow \Sigma, \overline{p}}$$

$$\frac{}{P, \Sigma \vdash_{\mathrm{p}} \varepsilon \rightsquigarrow \varepsilon, \varepsilon} \qquad \frac{P, \Sigma_0 \vdash_{\mathrm{d}} d \rightsquigarrow \Sigma_1, \overline{p}_1 \qquad P \vdash_{\mathrm{d}} d :: D \qquad (P; D), (\Sigma_0; \Sigma_1) \vdash_{\mathrm{p}} p \rightsquigarrow \Sigma_2, \overline{p}_2}{P, \Sigma_0 \vdash_{\mathrm{p}} (d; p) \rightsquigarrow (\Sigma_1; \Sigma_2), (\overline{p}_1; \overline{p}_2)}$$

$$\boxed{P, \Sigma \vdash_{\mathrm{d}} d \rightsquigarrow \Sigma, \overline{p}}$$

$$\frac{(\Sigma; i_1 \mapsto \mathsf{self}) \vdash_{\mathrm{e}} e_1 \rightsquigarrow \overline{e}_1 \qquad (\Sigma; i_1 \mapsto \mathsf{self}) \vdash_{\mathrm{e}} e_2 \rightsquigarrow \overline{e}_2 \qquad \overrightarrow{(\Sigma; i_1 \mapsto \mathsf{self}) \vdash_{\mathrm{h}} h \rightsquigarrow \overline{h}}}{P, \Sigma \vdash_{\mathrm{d}} i_0 = \mathsf{fun}\, i_1(\overrightarrow{i_2})\, e_1\, e_2\, \overrightarrow{h} \rightsquigarrow i_0 \mapsto \mathsf{self}, i_0 = \mathsf{fun}\, i_1(\overrightarrow{i_2})\, \overline{e}_1\, \overline{e}_2\, \overrightarrow{\overline{h}}}$$

$$\frac{\Sigma \vdash_{\mathrm{e}} e \rightsquigarrow \overline{e} \qquad \overrightarrow{\Sigma \vdash_{\mathrm{r}} r \rightsquigarrow \overline{r}} \qquad \overrightarrow{\Sigma \vdash_{\mathrm{h}} h \rightsquigarrow \overline{h}}}{P, \Sigma \vdash_{\mathrm{d}} i_0 = \mathsf{thm}\,(\overrightarrow{i_1})\, e\, \overrightarrow{r}\, \overrightarrow{h} \rightsquigarrow i_0 \mapsto \mathsf{self}, i_0 = \mathsf{thm}\,(\overrightarrow{i_1})\, \overline{e}\, \overrightarrow{\overline{r}}\, \overrightarrow{\overline{h}}}$$

$$\frac{P, \Sigma_1 \vdash_{\mathrm{t}} t \rightsquigarrow \Sigma_2, \sigma, \overline{p}}{P, \Sigma_1 \vdash_{\mathrm{d}} i = t \rightsquigarrow (\Sigma_2; i \mapsto \sigma), \overline{p}}$$

$$\boxed{P, \Sigma \vdash_{\mathrm{t}} t \rightsquigarrow \Sigma, \sigma, \overline{p}}$$

$$\frac{\Sigma \vdash_{\mathrm{a}} a \hookrightarrow i}{P, \Sigma \vdash_{\mathrm{t}} a \rightsquigarrow \varepsilon, \mathsf{alias}\, i, \varepsilon} \qquad \frac{\overrightarrow{i_2} \text{ fresh} \qquad P, \Sigma_1 \vdash_{\mathrm{p}} \overrightarrow{i_2 = [\overrightarrow{i_2}/\overrightarrow{i_1}]t} \rightsquigarrow \Sigma_2, \overline{p}}{P, \Sigma_1 \vdash_{\mathrm{t}} \mathsf{inst}\{\overrightarrow{\ell \triangleright i_1 = t}\} \rightsquigarrow \Sigma_2, \mathsf{inst}\,\{\overrightarrow{\ell \triangleright i_2}\}, \overline{p}}$$

$$\frac{P, \Sigma_0 \vdash_{\mathrm{T}} T \rightsquigarrow \Sigma_1, \sigma_1, \overline{p}_1 \qquad (P; i : T), (\Sigma_0; \Sigma_1; i \mapsto \sigma_1) \vdash_{\mathrm{t}} t \rightsquigarrow \Sigma_2, \sigma_2, \overline{p}_2}{P, \Sigma_0 \vdash_{\mathrm{t}} \mathsf{gen}\,(i : T)\, t \rightsquigarrow \varepsilon, \mathsf{none}, \mathsf{hidden}\,\{\mathsf{assume}\,\{\overline{p}_1\}; \overline{p}_2\}}$$

$$\frac{P, \Sigma_0 \vdash_{\mathrm{t}} t \rightsquigarrow \Sigma_1, \sigma_1, \overline{p}_1 \qquad P \vdash_{\mathrm{t}} t(a) :: T \qquad P, \Sigma_0 \vdash_{\mathrm{T}} T \rightsquigarrow \Sigma_2, \sigma_2, \overline{p}_2}{P, \Sigma_0 \vdash_{\mathrm{t}} t(a) \rightsquigarrow (\Sigma_1; \Sigma_2), \sigma_2, (\overline{p}_1; \mathsf{assume}\,\{\overline{p}_2\})}$$

$$\frac{P, \Sigma_0 \vdash_{\mathrm{t}} t \rightsquigarrow \Sigma_1, \sigma_1, \overline{p}_1 \qquad P, \Sigma_0 \vdash_{\mathrm{T}} T \rightsquigarrow \Sigma_2, \sigma_2, \overline{p}_2}{P, \Sigma_0 \vdash_{\mathrm{t}} \mathsf{seal}\, t : T \rightsquigarrow \Sigma_2, \sigma_2, (\mathsf{hidden}\,\{\overline{p}_1\}; \mathsf{assume}\,\{\overline{p}_2\})} \qquad \frac{}{P, \Sigma \vdash_{\mathrm{t}} \mathsf{type}\, T \rightsquigarrow \varepsilon, \mathsf{none}, \varepsilon}$$

Figure 5.18: Proof obligations for programs, definitions, and terms.

an environment and initial shape mapping.

Function and theorem definitions, however, do each correspond to a single ACL2 definition. Their constituent expressions, rule classes, and hints are each translated to ACL2 using the shape mapping, extended in the case of functions to include the function's local name. These translated components are used to construct an ACL2 definition; function and theorem names are always mapped to the shape self.

Other definitions are translated based on the verification obligation for their term. The resulting shape mappings and ACL2 definitions are produced for the whole definition. The shape mapping is extended to map the definition's name to the shape produced for its term.

The ACL2 form of terms is determined by the judgment $P, \Sigma_1 \vdash_{\mathrm{t}} t \rightsquigarrow \Sigma_2, \sigma, \overline{p}$. As with programs and definitions, terms are translated in the context of an environment and a shape mapping. The verification process for a term produces its own shape in addition to its ACL2 proof obligations and any additional shape mappings.

Addresses incur no proof obligations, as the term to which they refer must already have its own proof obligation. An address likewise introduces no new shape mappings; its own shape is an alias to the ACL2 name corresponding to the address.

We produce the verification obligation for an instance term by first assigning fresh names to each definition in the instance. By translating these renamed definitions, we obtain new shape mappings and proof obligations. The instance's proof obligation consists of these mappings and obligations plus an instance shape mapping labels to their new names.

The proof obligation for a generic consists of two key parts: an assumption of the logical consequences of the input type and the obligations of the body term. The body is translated using the environment and shape mapping produced by translating the input type. The result for the whole generic term introduces no shape mappings and has a shape of none, as the contents of a generic are not directly accessible. The generic's verification obligation is hidden from the rest of the ACL2 program, as its logical conclusions are not valid without a witness for its input. The consequences of the input are assumed, and from these the body of the generic can be verified.

Applying a generic introduces no additional proof obligations beyond those necessary for the generic itself. The consequences of the resulting instance type can therefore safely be assumed and used by the rest of the program.

When a term is sealed at a type, we hide the term's verification from the rest of the program. This essentially abstracts the details of the term's implementation out of subsequent proofs. The rest of the proof obligation assumes the consequences of the sealed type, giving a potentially simpler theory from which to reason.

Type components have neither proof obligations nor shape mappings of their own, and are assigned the shape none.

The rules for generating the consequences of environments, declarations, and types are given in figure 5.19. These rules closely mirror those for programs, definitions, and terms except that all hidden verification obligations are omitted.

Environments are translated to ACL2 one declaration at a time, accumulating bindings in the environment and shape mapping for each one. The empty environment produces an empty shape mapping and an empty proof obligation. This process is defined by the judgment $P_1, \Sigma_1 \vdash_{\mathrm{P}} P_2 \rightsquigarrow \Sigma_2, \overline{p}$.

We generate the logical consequences of each individual declaration via the relation

$$\boxed{P, \Sigma \vdash_{\mathrm{P}} P \rightsquigarrow \Sigma, \overline{p}}$$

$$\frac{}{P, \Sigma \vdash_{\mathrm{P}} \varepsilon \rightsquigarrow \varepsilon, \varepsilon} \qquad \frac{P_1, \Sigma_0 \vdash_{\mathrm{D}} D \rightsquigarrow \Sigma_1, \overline{p}_1 \qquad (P_1; D), (\Sigma_0; \Sigma_1) \vdash_{\mathrm{P}} P_2 \rightsquigarrow \Sigma_2, \overline{p}_2}{P_1, \Sigma_0 \vdash_{\mathrm{P}} (D; P_2) \rightsquigarrow (\Sigma_1; \Sigma_2), (\overline{p}_1; \overline{p}_2)}$$

$$\boxed{P, \Sigma \vdash_{\mathrm{D}} D \rightsquigarrow \Sigma, \overline{p}}$$

$$\frac{}{P, \Sigma \vdash_{\mathrm{D}} i : \mathsf{stub}\,(n) \rightsquigarrow i \mapsto \mathsf{self}, i : \mathsf{stub}\,(n)}$$

$$\frac{(\Sigma; i_1 \mapsto \mathsf{self}) \vdash_{\mathrm{e}} e_1 \rightsquigarrow \overline{e}_1 \qquad (\Sigma; i_1 \mapsto \mathsf{self}) \vdash_{\mathrm{e}} e_2 \rightsquigarrow \overline{e}_2}{P, \Sigma \vdash_{\mathrm{D}} i_0 : \mathsf{fun}\, i_1(\overrightarrow{i_2})\, e_1\, e_2 \rightsquigarrow i_0 \mapsto \mathsf{self}, i_0 : \mathsf{fun}\, i_1(\overrightarrow{i_2})\, \overline{e}_1\, \overline{e}_2}$$

$$\frac{\Sigma \vdash_{\mathrm{e}} e \rightsquigarrow \overline{e} \qquad \overrightarrow{\Sigma \vdash_{\mathrm{r}} r \rightsquigarrow \overline{r}}}{P, \Sigma \vdash_{\mathrm{d}} i_0 : \mathsf{thm}\,(\overrightarrow{i_1})\, e\, \overrightarrow{r} \rightsquigarrow i_0 \mapsto \mathsf{self}, i_0 : \mathsf{thm}\,(\overrightarrow{i_1})\, \overline{e}\, \overrightarrow{\overline{r}}} \qquad \frac{P, \Sigma_1 \vdash_{\mathrm{T}} T \rightsquigarrow \Sigma_2, \sigma, \overline{p}}{P, \Sigma_1 \vdash_{\mathrm{D}} i : T \rightsquigarrow (\Sigma_2; i \mapsto \sigma), \overline{p}}$$

$$\boxed{P, \Sigma \vdash_{\mathrm{T}} T \rightsquigarrow \Sigma, \sigma, \overline{p}}$$

$$\frac{P \vdash_{\mathrm{T}} T_1 \hookrightarrow T_2 \qquad P, \Sigma_1 \vdash_{\mathrm{T}} T_2 \rightsquigarrow \Sigma_2, \sigma, \overline{p}}{P, \Sigma_1 \vdash_{\mathrm{T}} T_1 \rightsquigarrow \Sigma_2, \sigma, \overline{p}} \qquad \frac{\Sigma \vdash_{\mathrm{a}} a \hookrightarrow i}{P, \Sigma \vdash_{\mathrm{T}} \mathsf{ref}\, a \rightsquigarrow \varepsilon, \mathsf{alias}\, i, \varepsilon}$$

$$\frac{\overrightarrow{i_2}\ \text{fresh} \qquad P, \Sigma_1 \vdash_{\mathrm{P}} \overrightarrow{i_2 : [\overrightarrow{i_2}/\overrightarrow{i_1}]T} \rightsquigarrow \Sigma_2, \overline{p}}{P, \Sigma_1 \vdash_{\mathrm{T}} \mathsf{inst}\{\overrightarrow{\ell \triangleright i_1 : T}\} \rightsquigarrow \Sigma_2, \mathsf{inst}\,\{\overrightarrow{\ell \triangleright i_2}\}, \overline{p}} \qquad \frac{}{P, \Sigma_0 \vdash_{\mathrm{T}} \mathsf{gen}\,(i : T_1)\ T_2 \rightsquigarrow \varepsilon, \mathsf{none}, \varepsilon}$$

$$\frac{}{P, \Sigma \vdash_{\mathrm{T}} \mathsf{type}\ T \rightsquigarrow \varepsilon, \mathsf{none}, \varepsilon}$$

Figure 5.19: Proof obligations for environments, declarations, and types.

$P, \Sigma_1 \vdash_{\mathrm{D}} D \rightsquigarrow \Sigma_2, \overline{p}$. Stubs, functions, and theorems are all transformed straightforwardly, merely resolving references in their expressions and rule classes and assigning their names a shape of self. Other declarations are translated based on their type.

Translating types to ACL2 relies on the judgment $P, \Sigma_1 \vdash_{\mathrm{T}} T \rightsquigarrow \Sigma_2, \sigma, \overline{p}$. We reduce types during this process, as address references and type refinements have no direct consequences of their own.

Reference types correspond to aliases. They have no logical consequences; they are merely mapped to their source identifier.

Instance types are given proof consequences based on their consituent declarations, which are assigned fresh names to avoid any possible name clashes in the ACL2 environment. The translation of these declarations forms the result for the instance type, with the addition of an instance shape mapping members to their freshly assigned names.

The types of generics and type components have no logical consequences.

Translation from Refined ACL2 hints, rule classes, and expressions to their ACL2 coun-

$$\boxed{\Sigma \vdash_{\mathrm{h}} h \rightsquigarrow \overline{h}}$$

$$\frac{\Sigma \vdash_{\mathrm{a}} a \hookrightarrow i \qquad \overrightarrow{\Sigma \vdash_{\mathrm{e}} e \rightsquigarrow \overline{e}}}{\Sigma \vdash_{\mathrm{h}} \mathsf{lemma}\ a(\overrightarrow{e}) \rightsquigarrow \mathsf{lemma}\ i(\overrightarrow{\overline{e}})} \qquad \frac{\Sigma \vdash_{\mathrm{e}} e \rightsquigarrow \overline{e}}{\Sigma \vdash_{\mathrm{h}} \mathsf{induct}\ e \rightsquigarrow \mathsf{induct}\ \overline{e}}$$

$$\boxed{\Sigma \vdash_{\mathrm{r}} r \rightsquigarrow \overline{r}}$$

$$\frac{\Sigma \vdash_{\mathrm{e}} e_1 \rightsquigarrow \overline{e}_1 \qquad \Sigma \vdash_{\mathrm{e}} e_2 \rightsquigarrow \overline{e}_2 \qquad \Sigma \vdash_{\mathrm{e}} e_3 \rightsquigarrow \overline{e}_3}{P \vdash_{\mathrm{r}} e_1\ \mathsf{equals}\ e_2\ \mathsf{when}\ e_3 \rightsquigarrow \overline{e}_1\ \mathsf{equals}\ \overline{e}_2\ \mathsf{when}\ \overline{e}_3} \qquad \frac{\Sigma \vdash_{\mathrm{e}} e_1 \rightsquigarrow \overline{e}_1 \qquad \Sigma \vdash_{\mathrm{e}} e_2 \rightsquigarrow \overline{e}_2}{P \vdash_{\mathrm{r}} e_1\ \mathsf{recurs\ as}\ e_2 \rightsquigarrow \overline{e}_1\ \mathsf{recurs\ as}\ \overline{e}_2}$$

$$\boxed{\Sigma \vdash_{\mathrm{e}} e \rightsquigarrow e}$$

$$\frac{}{\Sigma \vdash_{\mathrm{e}} i \rightsquigarrow i} \qquad \frac{}{\Sigma \vdash_{\mathrm{e}} n \rightsquigarrow n} \qquad \frac{\Sigma \vdash_{\mathrm{e}} e_1 \rightsquigarrow \overline{e}_1 \qquad \Sigma \vdash_{\mathrm{e}} e_2 \rightsquigarrow \overline{e}_2 \qquad \Sigma \vdash_{\mathrm{e}} e_3 \rightsquigarrow \overline{e}_3}{\Sigma \vdash_{\mathrm{e}} \mathsf{if}\ e_1\ e_2\ e_3 \rightsquigarrow \mathsf{if}\ \overline{e}_1\ \overline{e}_2\ \overline{e}_3}$$

$$\frac{\Sigma \vdash_{\mathrm{e}} e_1 \rightsquigarrow \overline{e}_1 \qquad \Sigma \vdash_{\mathrm{e}} e_2 \rightsquigarrow \overline{e}_2}{\Sigma \vdash_{\mathrm{e}} \mathsf{let}\ i = e_1\ \mathsf{in}\ e_2 \rightsquigarrow \mathsf{let}\ i = \overline{e}_1\ \mathsf{in}\ \overline{e}_2} \qquad \frac{\Sigma \vdash_{\mathrm{a}} a \hookrightarrow i \qquad \overrightarrow{\Sigma \vdash_{\mathrm{e}} e \rightsquigarrow \overline{e}}}{\Sigma \vdash_{\mathrm{e}} a(\overrightarrow{e}) \rightsquigarrow i(\overrightarrow{\overline{e}})}$$

$$\boxed{\Sigma \vdash_{\mathrm{a}} a \hookrightarrow i}$$

$$\frac{i \mapsto \mathsf{self} \in \Sigma}{\Sigma \vdash_{\mathrm{a}} i \hookrightarrow i} \qquad \frac{i \mapsto \mathsf{none} \in \Sigma}{\Sigma \vdash_{\mathrm{a}} i \hookrightarrow i} \qquad \frac{i_1 \mapsto \mathsf{alias}\ i_2 \in \Sigma \qquad \Sigma \vdash_{\mathrm{a}} i_2 \hookrightarrow i_3}{\Sigma \vdash_{\mathrm{a}} i_1 \hookrightarrow i_3} \qquad \frac{i_1 \mapsto \mathsf{inst}\ \{\overrightarrow{\ell \triangleright i_2}\}}{\Sigma \vdash_{\mathrm{a}} i_1 \hookrightarrow i_1}$$

$$\frac{\Sigma \vdash_{\mathrm{a}} a \hookrightarrow i_0 \qquad i_0 \mapsto \mathsf{inst}\ \{\overrightarrow{\ell_2 \triangleright i_2}; \ell_1 \triangleright i_1; \overrightarrow{\ell_3 \triangleright i_3}\} \in \Sigma \qquad \Sigma \vdash_{\mathrm{a}} i_1 \hookrightarrow i_4}{\Sigma \vdash_{\mathrm{a}} a \centerdot \ell_1 \hookrightarrow i_4}$$

Figure 5.20: Proof obligations for hints, rule classes, expressions, and addresses.

terparts proceeds recursively according to the judgments $\Sigma \vdash_{\mathrm{h}} h \rightsquigarrow \overline{h}$; $\Sigma \vdash_{\mathrm{r}} r \rightsquigarrow \overline{r}$; and $\Sigma \vdash_{\mathrm{e}} e \rightsquigarrow \overline{e}$. Applied function and lemma addresses are resolved to ACL2 identifiers; otherwise the result is the same as the original in all three cases. See figure 5.20 for details.

Addresses resolve to ACL2 identifiers based on their shape. Identifiers mapped to alias shapes are resolved by following the alias; all other identifiers resolve to themselves. When resolving a member reference, the base address must resolve to an identifier with an instance shape. The resolution process proceeds by resolving the the identifier listed for the appropriate label in the instance shape.

The verification obligation for the example program from figures 5.3 and 5.4 is shown in figures 5.21 and 5.22.

The proof obligation for TYPE and LISTOF is empty; types incur no obligations.

The entire proof obligation for Listof is hidden, as generics only contribute to the outside environment when applied. Inside the hidden block, the proof begins by assuming the consequences of E : TYPE. This establishes E.is? as a stub and states that E.is?/boolean?

```
Obligations of Listof:
hidden
{ Consequences of Listof's formal, E:
  assume
  { E.is? = stub (1);
    E.is?/boolean? =
      thm (x1)
        boolean?(E.is?(x1))
        boolean?(E.is?(x1)) equals #true when #true
        ε };
  Obligations of Listof's body:
  hidden
  { filter2 =
      fun filter3(xs2)
        if cons?(xs2)
          let x3 = first(xs2)
          in let xs3 = filter3(rest(xs2))
          in if E.is?(x3) cons(x3, xs3) xs3
          void()
        0
        ε;
    filter/member2 =
      thm (x4, xs4)
        implies(member(x4, filter1(xs4)), E.is?(x4))
        E.is?(x4) equals #true when #true
        ε };
  Consequences of Listof's body:
  assume
  { Listof.filter : stub (1);
    Listof.filter/member =
      thm (x2, xs1)
        implies(member(x2, filter1(xs1)), E.is?(x2))
        E.is?(x2) equals #true when #true
        ε } }
```

Figure 5.21: Proof obligation of example program, part 1.

holds.[1]

The body of Listof is a sealed term, so the verification of the nested instance term is hidden. This hidden block holds the contents of the original instance, with references to E.is? resolved to E.is?.

Outside the hidden block for the sealed term, the obligation for Listof's body concludes by assuming the consequences of the sealed type. These assumptions consist of the stub Listof.filter and the theorem Listof.filter/member.

[1] We use "dotted" names such as E.is? and E.is?/boolean? for readability. Any fresh name may be used for functions and theorems in proof obligations.

```
Obligations of Integer:
hidden
{ is?/boolean?2 =
    thm (x5)
      boolean?(integer?(x5))
      boolean?(integer?(x5)) equals #true when #true
      ε };
Consequences of Integer:
assume
{ Integer.is?/boolean? =
    thm (x1)
      boolean?(integer?(x1))
      boolean?(integer?(x1)) equals #true when #true
      ε };
Consequences of Listof-Integer:
assume
{ Listof-Integer.filter : stub (1);
  Listof-Integer.filter/member =
    thm (x2, xs1)
      implies(member(x2, filter1(xs1)), integer?(x2))
      integer?(x2) equals #true when #true
      ε }
```

Figure 5.22: Proof obligation of example program, part 2.

$$
\begin{array}{lll}
p_1; i = a; p_2 & \longrightarrow p_1; [a/i]p_2 & \\
p_1; i = \mathsf{type}\ T; p_2 & \longrightarrow p_1; p_2 & \text{where } i \notin \mathrm{fv}(p_2) \\
p_1; i = \mathsf{seal}\ t : T; p_2 & \longrightarrow p_1; i = t; p_2 & \\
p_1; i_1 = \mathsf{gen}\ (i_2 : T)\ t; p_2 & \longrightarrow p_1; [\mathsf{gen}\ (i_2 : T)\ t/i_1]p_2 & \\
p_1; i_1 = (\mathsf{gen}\ (i_2 : T)\ t)(a); p_2 & \longrightarrow p_1; i_1 = [a/i_2]t; p_2 & \\
p_1; i_1 = \mathsf{inst}\{\overrightarrow{\ell \triangleright i_2 = t}\}; p_2 & \longrightarrow p_1; i_3 = [\overrightarrow{i_3/i_2}]t; [\overrightarrow{i_3/i_1.\ell}]p_2 & \text{where } \overrightarrow{i_3} \text{ fresh and } i_1 \notin \mathrm{fr}(p_2)
\end{array}
$$

Figure 5.23: Executable translation for Refined ACL2.

Our next proof obligation is that of Integer. Like the body of Listof, this term is sealed. Its obligation consists of two parts: a hidden block for the contents of the instance, and an assumption block for the consequences of its sealed type.

Finally, the proof obligation of Listof-Integer is merely an assumption of its type's consequences. Since Listof is verified above, there is nothing more to prove. The assumptions introduce Listof-Integer.filter as a stub and state the property Listof-Integer.filter/member.

5.5 Executable Semantics

The executable semantics for Refined ACL2 is generated by the step relation shown in figure 5.23. The judgment $p_1 \longrightarrow p_2$ means that program p_1 takes a step resulting in p_2. Each step eliminates a component construct from the overall program. Ultimately, every sequence of steps results in a program that is equal to its own verification obligation.[2]

Alias definitions are eliminated by substituting the target address for all references to the name of the alias.

Type definitions are simply dropped from the program once no further references to them remain.

Definitions of sealed terms step to the unsealed version of the term; logical hiding does not affect the ultimate behavior of the program.

Generics are substituted for their name. Elsewhere we avoid substituting terms for identifiers, as many positions in the grammar accept only addresses. However, in a well-typed program, references to generics can only occur in positions where all terms are allowed.

We replace the application of a generic with its body, with the actual argument substituted for the formal argument.

The definitions inside instances are lifted to the top level under fresh names and all references to members of the instance are replaced with these names. This step may only occur when all references to the instance are used to look up a member by its label, as determined by the $fr\ldots$ metafunction. Other uses, such as arguments to generics, must be eliminated before this step.

Figure 5.24 shows the executable form of the program from figures 5.3 and 5.4. The following steps construct this program. Some steps may be transposed; the result is equivalent up to choice of fresh names. We first substitute the definition of Listof into its application in Listof-Integer. Next, we resolve the application by substitution. Subsequently, we unseal the resulting instance. Once it is unsealed, we lift the instance's definitions to the top level. By substituting integer? for its alias, we eliminate another definition. As with Listof-Integer, we unseal and lift the contents of Integer and resolve its alias definition. Finally, we drop the definitions of TYPE and LISTOF, as no references to them remain.

5.6 Soundness of Refined ACL2

We verify Refined ACL2 programs using their proof obligations, but we execute their fully-linked forms. In order to ensure that the process of verification tells us something meaningful

[2] We do not prove this property; our soundness proof does not rely on it.

```
Integer.is?/boolean? =
  thm (x5)
    boolean?(is?2(x5))
    boolean?(is?2(x5)) equals #true when #true
    ε;
Listof-Integer.filter =
  fun filter3(xs2)
    if cons?(xs2)
      let x3 = first(xs2)
      in let xs3 = filter3(rest(xs2))
        in if integer?(x3) cons(x3, xs3) xs3
      void()
    0
    ε;
Listof-Integer.filter/member =
  thm (x4, xs4)
    implies(member(x4, Listof-Integer.filter(xs4)), integer?(x4))
    integer?(x4) equals #true when #true
    ε
```

Figure 5.24: Executable form of example program.

about the program we execute, we must prove a soundness theorem. Our theorem states that when we verify the proof obligation of a well-typed program, then the executable form of the same program must be logically valid as well. This holds regardless of whether the verification process—specifically, the proof heuristics in the ACL2 theorem prover—would be able to verify the program directly in its fully-linked form.

In order to reason about the relationship between ACL2 types and Refined ACL2 types, we introduce a "type-flattening" relation; see figure 5.25. Given a shape mapping, we can translate a Refined ACL2 environment to an ACL2 environment by lifting the types of instance members to top-level declarations. The shape mapping provides names for these lifted declarations.

Theorem 5.6.1 (Soundness)**.** *If* $(\varepsilon \vdash_{\mathrm{p}} p_1 :: P_1)$*;* $(\varepsilon, \varepsilon \vdash_{\mathrm{p}} p_1 \rightsquigarrow \Sigma_1, \overline{p}_1)$*;* $(p_1 \longrightarrow p_2)$*;* $(\varepsilon, \varepsilon \vdash_{\mathrm{p}} p_2 \rightsquigarrow \Sigma_2, \overline{p}_2)$*; and* $(\varepsilon \models_{\overline{\mathrm{p}}} \overline{p}_1)$*; then* $(\varepsilon \models_{\overline{\mathrm{p}}} \overline{p}_2)$*.*

Proof. The proof of the soundness theorem proceeds by cases on the derivation of $p_1 \longrightarrow p_2$. In each case, p_1 is divided up into $p_{\mathrm{a}}; d; p_{\mathrm{b}}$.

- Case $d \coloneqq i = a$. In this case, $\overline{p}_1$ is equal to $\overline{p}_2$ by induction on the derivation of the verification obligation of $[a/i]p_{\mathrm{b}}$.

- Case $d \coloneqq i = \mathsf{type}\ T$ where i does not appear free in p_{b}. In this case, d is dropped. As in the case above, $\overline{p}_1$ is the same as $\overline{p}_2$. Once again, the proof proceeds based on

$$\boxed{P, \Sigma \vdash P \cong \overline{P}}$$

$$\frac{}{P, \Sigma \vdash \varepsilon \cong \varepsilon} \qquad \frac{P_1, \Sigma \vdash D \cong \overline{P}_1 \qquad P_1; D, \Sigma \vdash P_2 \cong \overline{P}_2}{P_1, \Sigma \vdash D; P_2 \cong \overline{P}_1; \overline{P}_2}$$

$$\boxed{P, \Sigma \vdash D \cong \overline{P}}$$

$$\frac{i_0 \mapsto \mathsf{self} \in \Sigma}{P, \Sigma \vdash \mathsf{stub}\,(n) \cong \mathsf{stub}\,(n)} \qquad \frac{i_0 \mapsto \mathsf{self} \in \Sigma \qquad \Sigma \vdash_{\mathrm{e}} e_1 \rightsquigarrow \overline{e}_1 \qquad \Sigma \vdash_{\mathrm{e}} e_2 \rightsquigarrow \overline{e}_2}{P, \Sigma \vdash i_0 : \mathsf{fun}\, i_1(\overrightarrow{i_2})\, e_1\, e_2 \cong i_0 : \mathsf{fun}\, i_1(\overrightarrow{i_2})\, \overline{e}_1\, \overline{e}_2}$$

$$\frac{i_0 \mapsto \mathsf{self} \in \Sigma \qquad \Sigma \vdash_{\mathrm{e}} e \rightsquigarrow \overline{e} \qquad \overrightarrow{\Sigma \vdash_{\mathrm{r}} r \rightsquigarrow \overline{r}}}{P, \Sigma \vdash i_0 : \mathsf{thm}\,(\overrightarrow{i_1})\, e\, \overrightarrow{r} \cong i_0 : \mathsf{thm}\,(\overrightarrow{i_1})\, \overline{e}\, \overrightarrow{\overline{r}}} \qquad \frac{i \mapsto \sigma \in \Sigma \qquad P, \Sigma \vdash T \mathbin{@} \sigma \cong \overline{P}}{P, \Sigma \vdash i : T \cong \overline{P}}$$

$$\boxed{P, \Sigma \vdash T \mathbin{@} \sigma \cong \overline{P}}$$

$$\frac{}{P, \Sigma \vdash T \mathbin{@} \mathsf{none} \cong \varepsilon} \qquad \frac{}{P, \Sigma \vdash T \mathbin{@} \mathsf{alias}\, i \cong \varepsilon}$$

$$\frac{P \vdash_{\mathrm{T}} T_1 \hookrightarrow \mathsf{inst}\{\overrightarrow{\ell \triangleright i_2 : T_2}\} \qquad P, \Sigma \vdash \overrightarrow{i_1 : [\overrightarrow{i_1}/\overrightarrow{i_2}]\, T_2} \cong \overline{P}}{P, \Sigma \vdash T_1 \mathbin{@} \mathsf{inst}\,\{\overrightarrow{\ell \triangleright i_1}\} \cong \overline{P}}$$

Figure 5.25: Type flattening relation.

the derivation of p_{b}'s verification obligation.

- Case $d = i_0 = \mathsf{gen}\,(i_1 : T)\, t$. In this case, the (hidden) verification obligation of the generic is duplicated once for every reference into which it is substituted, each time with different fresh names. These duplicate proofs naturally hold.

- Case $d = i = \mathsf{inst}\{\overrightarrow{\ell \triangleright d_1}\}$. The resulting verification obligation $\overline{p}_2$ is the same as $\overline{p}_1$; lifting the instance's contents to the top level is also performed by the verification translation. The proof proceeds by induction on the derivation of $\overline{p}_2$.

- Case $d \coloneqq i = \mathsf{seal}\, t : T$. We know that p_2 is equal to $p_{\mathrm{a}}; i = t; p_{\mathrm{b}}$. From the derivations of $\overline{p}_1$ and $\overline{p}_2$, we know the following:

$$\begin{array}{cc}
\varepsilon, \varepsilon \vdash_{\mathrm{p}} p_{\mathrm{a}} \rightsquigarrow \Sigma_{\mathrm{a}}, \overline{p}_{\mathrm{a}} & \varepsilon \vdash_{\mathrm{p}} p_{\mathrm{a}} :: P_{\mathrm{a}} \\
P_{\mathrm{a}} \vdash_{\mathrm{t}} t :: T_0 & P_{\mathrm{a}} \vdash_{\mathrm{T}} T_0 \leqslant T \\
P_{\mathrm{a}}, \Sigma_{\mathrm{a}} \vdash_{\mathrm{t}} t \rightsquigarrow \Sigma_{\mathrm{z}}, \sigma_{\mathrm{z}}, \overline{p}_{\mathrm{z}} & P_{\mathrm{a}}, \Sigma_{\mathrm{a}} \vdash_{\mathrm{T}} T \rightsquigarrow \Sigma_{\mathrm{x}}, \sigma_{\mathrm{x}}, \overline{p}_{\mathrm{x}} \\
P_{\mathrm{a}}; i : T \vdash_{\mathrm{p}} p_{\mathrm{b}} :: P_{\mathrm{b}} & P_{\mathrm{a}}; i : T, \Sigma_{\mathrm{a}}; \Sigma_{\mathrm{x}}; i \mapsto \sigma_{\mathrm{x}} \vdash_{\mathrm{p}} p_{\mathrm{b}} \rightsquigarrow \Sigma_{\mathrm{b}}, \overline{p}_{\mathrm{b}} \\
P_{\mathrm{a}}; i : T_0 \vdash_{\mathrm{p}} p_{\mathrm{b}} :: P_{\mathrm{c}} & P_{\mathrm{a}}; i : T_0, \Sigma_{\mathrm{a}}; \Sigma_{\mathrm{z}}; i \mapsto \sigma_{\mathrm{z}} \vdash_{\mathrm{p}} p_{\mathrm{b}} \rightsquigarrow \Sigma_{\mathrm{c}}, \overline{p}_{\mathrm{c}}
\end{array}$$

Lemma 5.6.7 states that a program's verification obligation is well-typed using the flattened version of the source program's environment. Using this lemma, we can

derive types for $\overline{p}_1$ and $\overline{p}_2$. We assign names to these types:

$$\begin{aligned}
\varepsilon &\vdash_{\overline{\mathrm{p}}} \overline{p}_{\mathrm{a}} :: \overline{P}_{\mathrm{a}} \\
\overline{P}_{\mathrm{a}} &\vdash_{\overline{\mathrm{p}}} \overline{p}_{\mathrm{x}} :: \overline{P}_{\mathrm{x}} \\
\overline{P}_{\mathrm{a}}; \overline{P}_{\mathrm{x}} &\vdash_{\overline{\mathrm{p}}} \overline{p}_{\mathrm{b}} :: \overline{P}_{\mathrm{b}} \\
\overline{P}_{\mathrm{a}} &\vdash_{\overline{\mathrm{p}}} \overline{p}_{\mathrm{z}} :: \overline{P}_{\mathrm{z}} \\
\overline{P}_{\mathrm{a}}; \overline{P}_{\mathrm{z}} &\vdash_{\overline{\mathrm{p}}} \overline{p}_{\mathrm{c}} :: \overline{P}_{\mathrm{c}}
\end{aligned}$$

According to the derivation of $\varepsilon \models_{\overline{\mathrm{p}}} \overline{p}_1$, we know that:

$$\begin{aligned}
\varepsilon &\models_{\overline{\mathrm{p}}} \overline{p}_{\mathrm{a}} \\
\overline{P}_{\mathrm{a}} &\models_{\overline{\mathrm{p}}} \overline{p}_{\mathrm{z}} \\
\overline{P}_{\mathrm{a}}; \overline{P}_{\mathrm{x}} &\models_{\overline{\mathrm{p}}} \overline{p}_{\mathrm{b}}
\end{aligned}$$

We must show that $\overline{P}_{\mathrm{a}}; \overline{P}_{\mathrm{z}} \models_{\overline{\mathrm{p}}} \overline{p}_{\mathrm{c}}$. We do this by introducing an intermediate step. In between the original proof obligation based on T and the unsealed obligation based on t, we introduce one based on T_0.

- $P_{\mathrm{a}}, \Sigma_{\mathrm{a}} \vdash_{\mathrm{T}} T_0 \rightsquigarrow \Sigma_{\mathrm{w}}, \sigma_{\mathrm{w}}, \overline{p}_{\mathrm{w}}$, which holds because the verification obligation relation is total on well-formed types. The proof of this proceeds by straightforward induction.
- $\overline{P}_{\mathrm{a}} \vdash_{\overline{\mathrm{p}}} \overline{p}_{\mathrm{w}} :: \overline{P}_{\mathrm{w}}$, by lemma 5.6.7.
- $P_{\mathrm{a}}; i = T_0, \Sigma_{\mathrm{a}}; \Sigma_{\mathrm{w}}; i \mapsto \sigma_{\mathrm{w}} \vdash_{\mathrm{p}} p_{\mathrm{b}} \rightsquigarrow \Sigma_{\mathrm{d}}, \overline{p}_{\mathrm{d}}$, which holds because the verification obligation translation is also total on well-typed programs. Once again, this is proved by induction on the type derivation for p_{b}.
- $\overline{P}_{\mathrm{a}}; \overline{P}_{\mathrm{w}} \models_{\overline{\mathrm{p}}} \overline{p}_{\mathrm{d}}$. By lemma 5.6.7, $\overline{P}_{\mathrm{x}}$ and $\overline{P}_{\mathrm{w}}$ are both derived from T_0, differing only based on Σ_{x} versus Σ_{z}. This amounts to a substitution of identifiers, by straightforward inspection of the rules for type flattening. Furthermore, $\overline{p}_{\mathrm{b}}$ and $\overline{p}_{\mathrm{d}}$ differ only by this renaming. Thus we can conclude that $\overline{P}_{\mathrm{a}}; \overline{P}_{\mathrm{w}} \models_{\overline{\mathrm{p}}} \overline{p}_{\mathrm{d}}$ by a trivial application of functional instantiation using the renaming induced by Σ_{x} versus Σ_{z}.

Finally, we must show that $\overline{P}_{\mathrm{a}}; \overline{P}_{\mathrm{z}} \models_{\overline{\mathrm{p}}} \overline{p}_{\mathrm{c}}$, which we do once again by functional instantiation. Lemma 5.6.11 tells us that because $P_{\mathrm{a}} \vdash_{\mathrm{T}} T_0 \leqslant T$, $\overline{P}_{\mathrm{z}}$ is also a subtype of $\overline{P}_{\mathrm{w}}$ based on the renaming induced by Σ_{z} versus Σ_{w}. Furthermore, $\overline{p}_{\mathrm{c}}$ differs from $\overline{p}_{\mathrm{d}}$ by the same renaming. Therefore functional instantiation can apply.

- Case $d = i_0 = \mathsf{gen}\,(i_1 : T_1)\,t(a)$. The type derivation establishes the following facts:

$$\begin{gathered}
\varepsilon \vdash_{\mathrm{p}} p_{\mathrm{a}} :: P_{\mathrm{a}} \\
P_{\mathrm{a}}; i_1 : T_1 \vdash_{\mathrm{t}} t :: T_2 \\
P_{\mathrm{a}} \vdash_{\mathrm{t}} a :: T_3 \\
P_{\mathrm{a}} \vdash_{\mathrm{T}} T_3 \leqslant T_1 \\
P_{\mathrm{a}}; i_0 : [a/_{i_1}]\,T_2 \vdash_{\mathrm{p}} p_{\mathrm{b}} :: P_{\mathrm{b}}
\end{gathered}$$

  By lemma 5.6.2, we can derive $P_{\mathrm{a}} \vdash_{\mathrm{t}} [a/_{i_1}]t :: [a/_{i_1}]\,T_2$.

  From the derivations of $\overline{p}_1$ and $\overline{p}_2$, we know the following:

$$\begin{gathered}
\varepsilon, \varepsilon \vdash_{\mathrm{p}} p_{\mathrm{a}} \rightsquigarrow \Sigma_{\mathrm{a}}, \overline{p}_{\mathrm{a}} \\
P_{\mathrm{a}}, \Sigma_{\mathrm{a}} \vdash_{\mathrm{T}} T_1 \rightsquigarrow \Sigma_{\mathrm{x}}, \sigma_{\mathrm{x}}, \overline{p}_{\mathrm{x}} \\
P_{\mathrm{a}}; i_1 : T_{\mathrm{a}}, \Sigma_{\mathrm{a}}; \Sigma_{\mathrm{x}}; i_1 \mapsto \sigma_{\mathrm{x}} \vdash_{\mathrm{t}} t \rightsquigarrow \Sigma_{\mathrm{y}}, \overline{p}_{\mathrm{y}}, \\
P_{\mathrm{a}}, \Sigma_{\mathrm{a}} \vdash_{\mathrm{T}} [a/_{i_1}]\,T_2 \rightsquigarrow \Sigma_{\mathrm{z}}, \sigma_{\mathrm{z}}, \overline{p}_{\mathrm{z}} \\
P_{\mathrm{a}}; i_0 : [a/_{i_1}]\,T_2, \Sigma_{\mathrm{a}}; \Sigma_{\mathrm{z}}; i_0 \mapsto \sigma_{\mathrm{z}} \vdash_{\mathrm{p}} p_{\mathrm{b}} \rightsquigarrow \Sigma_{\mathrm{b}}, \overline{p}_{\mathrm{b}} \\
P_{\mathrm{a}}, \Sigma_{\mathrm{a}} \vdash_{\mathrm{t}} [a/_{i_1}]t \rightsquigarrow \Sigma_{\mathrm{w}}, \sigma_{\mathrm{w}}, \overline{p}_{\mathrm{w}} \\
P_{\mathrm{a}}; i_0 : [a/_{i_1}]\,T_2, \Sigma_{\mathrm{a}}; \Sigma_{\mathrm{w}}; i_0 \mapsto \sigma_{\mathrm{w}} \vdash_{\mathrm{p}} p_{\mathrm{b}} \rightsquigarrow \Sigma_{\mathrm{c}}, \overline{p}_{\mathrm{c}}
\end{gathered}$$

  Once again, lemma 5.6.7 allows us to assign types to our verification obligations:

$$\begin{gathered}
\varepsilon \vdash_{\overline{\mathrm{p}}} \overline{p}_{\mathrm{a}} :: \overline{P}_{\mathrm{a}} \\
\overline{P}_{\mathrm{a}} \vdash_{\overline{\mathrm{p}}} \overline{p}_{\mathrm{x}} :: \overline{P}_{\mathrm{x}} \\
\overline{P}_{\mathrm{a}}; \overline{P}_{\mathrm{x}} \vdash_{\overline{\mathrm{p}}} \overline{p}_{\mathrm{y}} :: \overline{P}_{\mathrm{y}} \\
\overline{P}_{\mathrm{a}} \vdash_{\overline{\mathrm{p}}} \overline{p}_{\mathrm{z}} :: \overline{P}_{\mathrm{z}} \\
\overline{P}_{\mathrm{a}}; \overline{P}_{\mathrm{z}} \vdash_{\overline{\mathrm{p}}} \overline{p}_{\mathrm{b}} :: \overline{P}_{\mathrm{b}} \\
\overline{P}_{\mathrm{a}} \vdash_{\overline{\mathrm{p}}} \overline{p}_{\mathrm{w}} :: \overline{P}_{\mathrm{w}} \\
\overline{P}_{\mathrm{a}}; \overline{P}_{\mathrm{w}} \vdash_{\overline{\mathrm{p}}} \overline{p}_{\mathrm{c}} :: \overline{P}_{\mathrm{c}}
\end{gathered}$$

  By the derivation of $\varepsilon \models_{\overline{\mathrm{p}}} \overline{p}_1$, we know:

$$\begin{gathered}
\varepsilon \models_{\overline{\mathrm{p}}} \overline{p}_{\mathrm{a}} \\
\overline{P}_{\mathrm{a}}; \overline{P}_{\mathrm{x}} \models_{\overline{\mathrm{p}}} \overline{p}_{\mathrm{y}} \\
\overline{P}_{\mathrm{a}}; \overline{P}_{\mathrm{z}} \models_{\overline{\mathrm{p}}} \overline{p}_{\mathrm{b}}
\end{gathered}$$

  From this we must show that $\overline{P}_{\mathrm{a}} \models_{\overline{\mathrm{p}}} \overline{p}_{\mathrm{w}}$ and $\overline{P}_{\mathrm{a}}; \overline{P}_{\mathrm{w}} \models_{\overline{\mathrm{p}}} \overline{p}_{\mathrm{c}}$. We show the former by functional instantiation based on the proof of $\overline{p}_{\mathrm{y}}$ and the renaming induced by substituting a for i_1 in $\Sigma_{\mathrm{a}}; \Sigma_{\mathrm{x}}; i_1 \mapsto \sigma_{\mathrm{x}}$. We show the latter by functional instantiation as well, based on the renaming mapping $\overline{P}_{\mathrm{z}}$ to $\overline{P}_{\mathrm{w}}$ and $\overline{p}_{\mathrm{b}}$ to $\overline{p}_{\mathrm{c}}$.

  □

Lemma 5.6.2 (Substitution and Term Types)**.** *If* $P_1 \vdash_{\mathrm{t}} a_1 :: T_1$, $P_1 \vdash_{\mathrm{T}} T_1 \leqslant T_2$, *and* $P_1; i : T_2; P_2 \vdash_{\mathrm{t}} t :: T_3$, *then* $P_1; [a_1/i]P_2 \vdash_{\mathrm{t}} [a_1/i]t :: T_4$ *such that* $P_1; [a_1/i]P_2 \vdash_{\mathrm{T}} T_4 \leqslant [a_1/i]T_3$.

Lemma 5.6.3 (Substitution and Type Well-Formedness)**.** *If* $P_1 \vdash_{\mathrm{t}} a_1 :: T_1$, $P_1 \vdash_{\mathrm{T}} T_1 \leqslant T_2$, *and* $P_1; i : T_2; P_2 \vdash_{\mathrm{T}} T_3$, *then* $P_1; [a_1/i]P_2 \vdash_{\mathrm{T}} [a_1/i]T_3$.

Proof. The proof of lemmas 5.6.2 and 5.6.3 proceeds by mutual induction on the derivation of the type of t and the well-formedness of T_3. We first consider the cases for the type of t.

- Case $t = \mathsf{inst}\{\overrightarrow{\ell}\ d\}$. The proof proceeds by sub-induction on the types assigned to $\overrightarrow{d}$. Considering the types for definitions adds cases for function and theorem definitions. These cases in turn hold by straightforward induction on the well-formedness derivation for hints, rule classes, and expressions.

- Case $t = \mathsf{gen}\,(i_0 : T_0)\,t_0$. The proof relies on the inductive hypothesis for the type of t_0 in the context where i_0 has the type $[a_1/i]T_0$.

- Case $t = t_1(a_2)$. Here we must know that subtype relationships are preserved by substitution, which follows from lemma 5.6.6. We conclude by using the inductive hypothesis for t_1.

- Case $t = \mathsf{seal}\,t_0 : T_0$. This case exploits the inductive hypotheses for the type of t_0 and for the well-formedness of T_0, as well as lemma 5.6.6 for subtype relationships.

- Case $t = \mathsf{type}\,T_0$. Once again, this case relies on the inductive hypothesis for the well-formedness of T_0.

- Case $t = a_2$. This case relies on lemmas 5.6.5 and 5.6.4 stating that address types and type reduction, respectively, are preserved under substitution.

We next consider the cases for the well-formedness of T_3.

- Case $T_3 = a_2$. In this case, we use lemmas 5.6.5 and 5.6.4 and the fact that encapsulated types are invariant under subtyping.

- Case $T_3 = T_4\,\mathsf{where}\,\overrightarrow{\ell} = a_2$. The inductive hypothesis tells us that T_4 is well-formed; we rely on lemmas 5.6.5 and 5.6.6 to show that a_2 still has an appropriate type.

- The remaining cases are either trivial or reduce directly to the inductive hypothesis.

□

Lemma 5.6.4 (Substitution and Type Reduction)**.** *If* $P_1 \vdash_{\mathrm{t}} a_1 :: T_1$*,* $P_1 \vdash_{\mathrm{T}} T_1 \leqslant T_2$*, and* $P_1; i_1 : T_2; P_2 \vdash_{\mathrm{T}} T_3 \hookrightarrow T_4$*, then* $P_1; [a_1/i_1]P_2 \vdash_{\mathrm{T}} [a_1/i_1]T_3 \hookrightarrow [a_1/i_1]T_4$*.*

Lemma 5.6.5 (Substitution and Environment Lookup)**.** *If* $P_1 \vdash_{\mathrm{t}} a_1 :: T_1$*,* $P_1 \vdash_{\mathrm{T}} T_1 \leqslant T_2$*, and* $P_1; i_1 : T_2; P_2 \vdash_{\mathrm{a}} a_2 :: T_3$*, then* $P_1; [a_1/i_1]P_2 \vdash_{\mathrm{a}} [a/i]a_2 :: T_4$ *such that* $P_1; [a_1/i_1]P_2 \vdash_{\mathrm{T}} T_4 \,\mathsf{where}\, \varepsilon = [a_1/i_1]a_2 \leqslant [a_1/i_1]T_3$*.*

Proof. The proof of lemmas 5.6.4 and 5.6.5 proceeds by mutual induction on the derivation of type reduction for T_3 and on the derivation of type lookup for a_2. We start by examining the cases for type reduction.

- Case $T_3 = a_2$. In this case, a_2 has type T_5 which reduces to $\mathsf{type}\, T_6$. Thus $T_4 = T_6$. By lemma 5.6.5, after substitution a_2 reduces to a subtype of T_5, also post-substitution. We can inspect the rules for subtypes to see that $\mathsf{type}\, T_6$ is only a subtype of itself. Therefore $[a_1/i_1]T_4$ is equal to $[a_1/i_1]T_6$, and the case reduces to the inductive hypothesis.
- Case $T_3 = \mathsf{inst}\{\overrightarrow{\ell \triangleright D}\}$. This proof proceeds by induction on the reduction of $\overrightarrow{D}$ and by the inductive hypothesis for each contained type.
- Case $T_3 = \mathsf{gen}\,(i_2 : T_5)\; T_6$. This proof proceeds by two applications of the inductive hypothesis.
- Case $T_3 = T_5 \,\mathsf{where}\, \overrightarrow{\ell_2} = a_2$. This proof proceeds by further cases on the form of T_5; each case is straightforward.
- Case $T_3 = \mathsf{ref}\; a_2$. This case relies on lemma 5.6.5.
- Case $T_3 = \mathsf{type}\; T_5$, holds by the inductive hypothesis.
- Transitive case: this case holds by two applications of the inductive hypothesis.
- Reflexive case: holds trivially.

Next, we examine the cases for looking up a_2 in the environment.

- Case $a_2 = i_2$.

  If $i_2 = i_1$, then $[a_1/i_1]a_2 = a_1$. Thus $T_1 = T_4 \,\mathsf{where}\, \varepsilon = a_1$, $T_2 = T_5$, and the conclusion naturally holds.

  If $i_2 \in \mathrm{dom}(P_1)$, then substitution has no effect on a_2 or its type as i_1 is not in scope for them.

If $i_2 \in \mathrm{dom}(P_2)$, then $[a_1/i_1]a_2 = i_2$ and $T_4 = [a_1/i_1]T_3$. This case also holds, as $T_3 \,\mathsf{where}\, \varepsilon = i_2$ is always a subtype of T_3. This is proved by straightforward induction on the derivation of type reduction for refined types.

- Case $a_2 = a_3 \,.\, \ell_1$.

  In this case, $P_1; i_1 : T_2; P_2 \vdash_\mathrm{a} a_3 :: T_5$ and $P_1; i_1 : T_2; P_2 \vdash_\mathrm{T} T_5 \hookrightarrow \mathsf{inst}\{\overrightarrow{\ell_2 \triangleright i_2 : T_6}$ $\ell_1 \triangleright i_3 : T_7$ $\overrightarrow{\ell_3 \triangleright i_4 : T_8}\}$. We thus know that $T_3 = [\overrightarrow{a_3.\ell_2/i_2}]T_7$. By the inductive hypothesis, we know that $P_1; [a_1/i_1]P_2 \vdash_\mathrm{a} [a/i]a_3 :: T_9$ such that $P_1; [a_1/i_1]P_2 \vdash_\mathrm{T} T_9 \,\mathsf{where}\, \varepsilon = [a_1/i_1]a_3 \leqslant [a_1/i_1]T_5$. The derivations of term reduction and substitution tell us that T_9 must therefore reduce to some instance type which, when refined to $[a_1/i_1]a_3$, is a subtype of the instance type which T_5 reduces to. By the derivation of subtypes for instance types, we can determine that the respective labels' types must therefore also be subtypes. The extra type bindings for instance members are irrelevant, as reduction for refined instance types removes all references to them. This case therefore holds.

□

Lemma 5.6.6 (Substitution and Subtyping)**.** *If* $P_1 \vdash_\mathrm{t} a_1 :: T_1$, $P_1 \vdash_\mathrm{T} T_1 \leqslant T_2$, *and* $P_1; i_1 : T_2; P_2 \vdash_\mathrm{T} T_3 \leqslant T_4$, *then* $P_1; [a_1/i_1]P_2 \vdash_\mathrm{T} [a_1/i_1]T_3 \leqslant [a_1/i_1]T_4$.

Proof. This proof is by induction on the derivation of subtyping of T_3 and T_4.

- Reduction case: holds by lemma 5.6.4.
- Reflexive case: holds trivially.
- Case $T_3 = \mathsf{fun}\, i_2(\overrightarrow{i_3})\, e_1\, e_2$ and $T_4 = \mathsf{stub}\,(|\overrightarrow{i_3}|)$. This case is trivial.
- Case $T_3 = \mathsf{fun}\, i_2(\overrightarrow{i_3})\, e_1\, e_2$ and $T_4 = \mathsf{fun}\, i_4(\overrightarrow{i_5})\, e_3\, e_4$. This case holds based on the preservation of expression equivalence under substitution. The proof proceeds by sub-induction on the derivation of expression equivalence.
- Case $T_3 = \mathsf{thm}\,(\overrightarrow{i_2})\, e_1\, \overrightarrow{r_1}$ and $T_4 = \mathsf{thm}\,(\overrightarrow{i_3})\, e_2\, \overrightarrow{r_2}$. This case holds based on the preservation of expression and rule class equivalence. Again, the proof proceeds by cases on rule classes and the prior proof that expression equivalence is preserved.
- Case $T_3 = \mathsf{inst}\{\overrightarrow{\ell \triangleright D}\}$. This case proceeds by induction on $\overrightarrow{D}$ and the inductive hypothesis for each contained type.
- Case $T_3 = \mathsf{gen}\,(i_2 : T_5)\, T_6$. This case uses the inductive hypothesis twice.

- Case $T_3 = \mathsf{ref}\ a_2$. The proof uses lemma 5.6.5 and the fact that for any well-formed type of the form $\mathsf{ref}\ a$, any well-formed type of the form $T\,\mathsf{where}\,\varepsilon = a$ must reduce to $\mathsf{ref}\ a$.

- Transitive case: holds by the inductive hypothesis twice.

□

Lemma 5.6.7 (Types for Verification Obligations of Programs)**.** *If* $\varepsilon, \Sigma_1 \vdash P_1 \cong \overline{P}_1$*,* $P_1 \vdash_{\mathrm{p}} p :: P_2$*,* $P_1, \Sigma_1 \vdash_{\mathrm{p}} p \rightsquigarrow \Sigma_2, \overline{p}$*, and* $P_1, \Sigma_1; \Sigma_2 \vdash P_2 \cong \overline{P}_2$*, then* $\overline{P}_1 \vdash_{\overline{\mathrm{p}}} \overline{p} :: \overline{P}_2$*.*

Lemma 5.6.8 (Types for Verification Obligations of Terms)**.** *If* $\varepsilon, \Sigma_1 \vdash P_1 \cong \overline{P}_1$*,* $P_1 \vdash_{\mathrm{t}} t_1 :: T_1$*,* $P_1, \Sigma_1 \vdash_{\mathrm{t}} t_1 \rightsquigarrow \Sigma_2, \sigma_1, \overline{p}$*, and* $P_1, \Sigma_1; \Sigma_2 \vdash T_1 @ \sigma_1 \cong \overline{P}_2$*, then* $\overline{P}_1 \vdash_{\mathrm{p}} \overline{p} :: \overline{P}_2$*.*

Proof. The proof of lemmas 5.6.7 and 5.6.8 proceeds by mutual induction on the two derivations of $\overline{p}$. We first examine the different kinds of definitions.

- Function definition case. This case proceeds by sub-induction on the translation of expressions.

- Theorem definition case. This case relies on the previous proof about expressions and by cases on rule classes.

- The final case proceeds by the inductive hypothesis for terms.

Next, we proceed by cases on terms.

- Reference case: trivial.

- Instance case: proceeds by the inductive hypothesis for programs.

- Generic case: proceeds by the inductive hypothesis for input and output type.

- Instantiation case: proceeds using the inductive hypothesis for the applied generic, and by lemma 5.6.10 for the formal's type.

- Sealing case: as above case, proceeds by the inductive hypothesis for its subterm and lemma 5.6.10 for the sealed type.

- Type case: trivial.

□

Lemma 5.6.9 (Types for Verification Consequences of Environments)**.** *If* $\varepsilon, \Sigma_1 \vdash P_1 \cong \overline{P}_1$*,* $P_1 \vdash_{\mathrm{P}} P_2$*,* $P_1, \Sigma_1 \vdash_{\mathrm{P}} P_2 \rightsquigarrow \Sigma_2, \overline{p}$*, and* $P_1, \Sigma_1; \Sigma_2 \vdash P_2 \cong \overline{P}_2$*, then* $\overline{P}_1 \vdash_{\overline{\mathrm{p}}} \overline{p} :: \overline{P}_2$*.*

Lemma 5.6.10 (Types for Verification Consequences of Types)**.** *If* $\varepsilon, \Sigma_1 \vdash P_1 \cong \overline{P}_1$*,* $P_1 \vdash_{\mathrm{T}} T_1$*,* $P_1, \Sigma_1 \vdash_{\mathrm{T}} T_1 \rightsquigarrow \Sigma_2, \sigma_1, \overline{p}$*, and* $P_1, \Sigma_1; \Sigma_2 \vdash T_1 @ \sigma_1 \cong \overline{P}_2$*, then* $\overline{P}_1 \vdash_{\mathrm{p}} \overline{p} :: \overline{P}_2$*.*

Proof. These two lemmas proceed symmetrically with lemmas 5.6.7 and 5.6.8, without instantiation or sealing cases but with a type reduction case that proceeds via the inductive hypothesis. □

Lemma 5.6.11 (Verification Obligations and Subtyping)**.** *If* $\varepsilon, \Sigma_0 \vdash P_0 \cong \overline{P}_0$*,* $P_0 \vdash_{\mathrm{T}} T_1 \leqslant T_2$*,* $P_0, \Sigma_0 \vdash_{\mathrm{T}} T_1 \rightsquigarrow \Sigma_1, \sigma_1, \overline{p}_1$*,* $P_0, \Sigma_0 \vdash_{\mathrm{T}} T_2 \rightsquigarrow \Sigma_2, \sigma_2, \overline{p}_2$*,* $P_0, \Sigma_0; \Sigma_1 \vdash T_1 @ \sigma_1 \cong \overline{P}_1$*, and* $P_0, \Sigma_0; \Sigma_2 \vdash T_2 @ \sigma_2 \cong \overline{P}_2$*, then* Σ_1 *and* σ_1 *versus* Σ_2 *and* σ_2 *induce a mapping from the domain of* $\overline{P}_2$ *to* $\overrightarrow{i_1}$*, and* $\overline{P}_0 \vdash_{\overline{\mathrm{P}}} \overline{P}_1 \leqslant \overline{P}_2 @ \overrightarrow{i_1}$*.*

Proof. The proof proceeds based on the derivation of subtyping.

- Type reduction case: by induction on the derivation of type reduction, we show that the verification artifacts are unchanged.
- Reflexive case: trivial.
- Function/stub case: trivial as well.
- Function and theorem cases: by induction on expression equivalence.
- Instance case: by induction on the sequence of member declarations and the I.H. for each one's type.
- Generic, type, and reference cases: trivial due to empty obligation.
- Transitive case: this case holds by the transitivity of functional instantiation.

□

5.7 Implementation Details

Refined ACL2 is implemented under the name Dracula using the language-extension features of Racket and the theorem proving engine of ACL2. Programs are written as Racket modules beginning with #lang dracula. Racket's compiler then performs macro expansion, compiling the program down to a simple core language. The Dracula implementation processes this compiled program, interpreting specific patterns of Racket primitives as elements of the Refined ACL2 grammar. The resulting Refined ACL2 is typechecked before continuing; if the program is ill-typed, Dracula reports an error and halts compilation. From a well-typed Refined ACL2 interpretation, Dracula produces an ACL2 book representing the proof

obligation of the program. Users can thus execute the compiled form of the Dracula program via Racket and verify its proof obligation using ACL2.

When compiling programs to an executable form, Dracula uses compound values for instances rather than lifting their members to the top level, and uses closures for generics rather than syntactically substituting their bodies at all application sites. This translation is designed to behave the same as our semantics from figure 5.23, while generally performing better during both compile-time and run-time.

In order to interpret fully-expanded Racket programs as Refined ACL2, we must recognize certain combinations of Racket primitives as Refined ACL2 forms. Furthermore, we must ensure that every Dracula program expands to one of these recognizable patterns. Some Racket primitives have multiple interpretations. For instance, the #%plain-lambda form—Racket's primitive function-creation mechanism—is used for Refined ACL2 functions, theorems, and generics in both terms and types. Others are not used at all, such as begin0. Dracula introduces "landmark" functions—e.g. make-generic to wrap #%plain-lambda, indicating its intended use—and other conventions to make parsing Racket programs as Refined ACL2 into a deterministic process.

At a higher level, Dracula comes equipped with a set of macros implementing function, theorem, and component definition forms such as those in figure 5.1. These macros must ensure that their full expansion can be interpreted as Refined ACL2. Furthermore, they must support the addition of macro definitions to instance types; these macros must be context sensitive, mapping local member references to the appropriate concrete instance when applied. We adapt the approach used for macros in Racket's unit system [Culpepper et al. 2005].

When constructing proof obligations, Dracula must map the value literals and primitives of Racket to those of ACL2. For literals, we use a fixed mapping for a core set of values. Immutable cons-pairs map to ACL2 pairs. Immutable strings using only ASCII characters map to ACL2 strings. ASCII characters map to ACL2 characters. Racket's exact numbers map to ACL2 numbers. Interned symbols in Racket map to ACL2 symbols in the "DRACULA" package; keywords map to ACL2 symbols in the "KEYWORD" package. ACL2 symbols in other packages have no Racket equivalent; similarly other Racket values such as mutable strings, inexact numbers, and user-defined structures have no ACL2 equivalent. Quoted literals in Dracula source programs are translated using this mapping to quoted ACL2 literals.

Primitive functions such as cons, car, and cdr are registered at compile time in Dracula along with an arity, package name, and symbol name for use in ACL2. For instance, cons? has arity 1 and maps to ACL2::CONSP. These mappings are used when creating proof

obligations when the registered functions are applied.

Of course, the grammar shown in figure 5.2 is not precisely what the ACL2 theorem prover uses. For instance, although local names for functions are convenient for our purposes, ACL2 has only one name for each function. We map function definitions from Refined ACL2 to **defun** forms in ACL2 by substituting the function's global name for its local name in the body, measure, and hints. Theorem definitions become **defthm** forms and stub definitions use **defstub**. An assumed sequence of definitions $\overrightarrow{d}$ translates to (**skip-proofs** (progn $\overline{d}$ ...)). Hidden sequences become (encapsulate () (local $\overline{d}$) ...). By using only two special forms of encapsulate—one where all definitions are local and the **defstub** macro—and no **defaxiom** forms, we avoid much of the complexity of the ACL2 logic in our formalism and translation.

5.8 Related Work

The design space of modules with explicit specifications has been well-explored. The literature begins with Modula-2 [Wirth 1983b], and includes more recent developments such as the ML module system [Harper and Lillibridge 1994; Leroy 1994], Racket units [Flatt and Felleisen 1998a; Owens and Flatt 2006b], and mixin modules [Dreyer and Rossberg 2008]. 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 modular constructs for theorem provers include Isabelle's locales [Kammüller et al. 1999], Coq's sections [The Coq Development Team 2006], and the "little theories" of IMPS [Farmer et al. 1993]. These constructs provide 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 constructs do not extend its expressivity, but instead provide syntactic convenience.

Extended ML (EML) [Sannella 1991] equips SML [Milner et al. 1990] with logical properties and a verification semantics. The language is designed around the top-down 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 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 [Courant 2007; Chrzaszcz 2003] inherits aspects of the ML module system and enriches it with a language of logical specifications. The two are sufficiently expressive 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 systems and builds on them. We use nestable modules with external linking and external interfaces with translucent, refineable member specifications as in ML functors and signatures. Refined 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 support higher-order abstractions. We have to synthesize a method for expressing, verifying, and instantiating abstract proofs within a first-order logic.

Refined 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.

Chapter 6

Finale

I set out to establish the following thesis:

> The language of ACL2 can be extended to express robust modular and syntactic abstractions without changing ACL2's logic or theorem prover.

Specifically, it is my intention to address several absent features of ACL2 that facilitate large-scale, expressive proof development.

Top-down reasoning is a natural way to develop a proof, but not one that ACL2 directly supports. In ACL2, a top-down proof requires a combination of axioms, stubs, and "skipped" proofs as placeholders for low-level details. These artifacts must be replaced with a fully implemented and verified version to complete the proof, which in turn invalidates the certification of the main theorem. In Refined ACL2, individual components can be verified and certified separately, then linked together in arbitrary order later without recertification.

Second-order abstraction—universal quantification—over proofs and functions is a powerful tool for developing multiple proofs with common elements. Writing and verifying key parts once avoids extra time spent on compilation, verification, and development, as well as reducing the chance for errors. ACL2's existing methods of abstracting over proofs and functions include encapsulation, which requires an initial witness and sacrifices execution, and macros, which only abstract syntactically and must be verified at each application site. Refined ACL2 provides a method to build and verify second-order abstractions in advance, then instantiate them later for execution and instance-specific reasoning.

For any kind of abstraction, a language of specifications for inputs and outputs can improve both conciseness and expressiveness. ACL2's encapsulated abstractions do not separate their specification from their implementation; producing two similar abstractions or two uses of one abstraction involves repeated effort. Instance types in Refined ACL2 can be defined, nested, reused, and refined as needed for a given application.

Being able to execute the logical model used to verify a program is an important capability. Execution permits the developer to compare a model to the original program and

to find counterexamples to failed theorems. ACL2 abstracts details away from parts of a program with encapsulation, which excludes execution.[1] Sealed components in Refined ACL2 expose none of their internal details, yet the overall program remains executable.

Managing names in ACL2 can be tricky at best. At the file level, the package system provides only weak guarantees about namespace separation, and little ability to import names. ACL2's unhygienic macros can also lead to problems with unintended name clashes. Refined ACL2 has a component system based on lexical scope, with imports based on explicit specifications. Overall, the language inherits Racket's expressive module and macro systems.

Using only the theorem proving capabilities already present in ACL2, Refined ACL2 provides all the development features we identify as missing in the original language. Our soundness proof establishes a trusted relationship between source programs and the theorems proved about them; long-standing experience and the reputations of Racket and ML establish the usefulness of their macro and module facilities, respectively.

In addition to satisfying our thesis, Refined ACL2 provides an ACL2 model for a limited subset of Racket programs. The extensible set of axiomatized primitives opens the door for modeling more of Racket as needed. With further research it may be possible to model more of Racket in ACL2, such as certain kinds of side effects, higher order programming, or perhaps more modest additions such as functional hash tables and regular expressions.

[1] Concurrently with this work, the ACL2 maintainers have added a feature called **defattach** which can restore executability to encapsulated code.

Appendix A

Modular ACL2: First Model

This appendix presents a model used in Modular ACL2's early design stages[1]. Section A.1 describes the language and section A.2 presents an overview of our two-part model. Section A.3 discusses the part of our model concerned with local reasoning about modules. Section A.4 explains the other half of our model, concerned with the execution of whole programs. Section A.5 evaluates the current design, addressing alternate options and implementation concerns.

A.1 Design of Modular ACL2

Our goal is to create a coherent linguistic construct for modular programming in ACL2. To that end we have designed the Modular ACL2 language using PLT Redex models. Programs in Modular ACL2 consist of *interfaces* and *modules*. Modules are program fragments with their own namespaces and logical theories; interfaces determine how the program and proof fragments from different modules interact.

In the current design of Modular ACL2, modules and interfaces are defined at the top level. Interfaces contain a set of function signatures and logical statements about them. Modules may be *primitive* or *compound*. Primitive modules import and export definitions via interfaces and contain internal definitions. Compound modules link other modules together, filling in the exports of one module as the implementation for the matching imports of another module. From the logical perspective, a module may be viewed as an implication, with the theorems of its imported interfaces as hypotheses and the theorems of its exported interfaces as conclusions. Linking a compound module discharges some of the hypotheses.

Figure A.1 demonstrates a sample Modular ACL2 program representing one possible translation of SQR-ALL. The program consists of two interfaces, three modules, and an entry point. The original functions are split into separate modules. The function `sqr` is

[1] This work has been published in Felleisen et al. [2009].

```
(interface INT-IFC
  (function int-fn (n) t)
  (theorem int-fn-type
    (implies (integerp n)
             (integerp (int-fn n)))))

(interface LIST-IFC
  (function list-fn (ns) t)
  (theorem list-fn-type
    (implies (integer-listp ns)
             (integer-listp (list-fn ns)))))

(module SQR-MOD
  (defun sqr (n) (* n n))
  (export INT-IFC (sqr as int-fn)))

(module MAP-MOD
  (import INT-IFC)
  (defun map-int-fn (ns)
    (cond ((endp ns) nil)
          ((consp ns) (cons (int-fn (car ns))
                            (map-int-fn (cdr ns))))))
  (export LIST-IFC (map-int-fn as list-fn)))

(compound SQR-ALL-MOD = MAP-MOD (SQR-MOD : INT-IFC))

(invoke LIST-IFC (list-fn (list 1 2 3)))
```

Figure A.1: The Modular ACL2 program SQR-ALL.

in the module `SQR-MOD`, which exports it as `int-fn` via interface `INT-IFC`. The theorem `int-fn-type` in the interface requires `int-fn` to be closed on the integers; when applied to `sqr`, it essentially restates the original conjecture `sqr-type`.

The other function, `sqr-all`, is split across multiple modules. The module `MAP-MOD` contains a generalized implementation which maps an arbitrary integer function (`int-fn` imported from `INT-IFC`) over a list. This function is exported as `list-fn` via `LIST-IFC`, which requires it to be closed on integer lists, as in `sqr-all-type`. The last module `SQR-ALL-MOD` links `MAP-MOD` to `SQR-MOD`, essentially reconstituting `sqr-all` by attaching the imported `int-fn` in `MAP-MOD` to `sqr` in `SQR-MOD`.

The program's entry point invokes the function `list-fn` exported by `SQR-ALL-MOD` on `(list 1 2 3)`.

The modular version of SQR-ALL splits the program into several reusable pieces, and protects each piece from interfering with the others. For instance, `SQR-MOD` does not have imports and does not export the name `sqr`, so the internal function cannot clash with another `sqr`. Also, the imported interface `INT-IFC` in `MAP-MOD` serves as an abstraction

```
program ::= (def ... expr ...)
    def ::= (defun fvar (var ...) expr)
          | (defthm fvar expr)
          | (defstub fvar (var ...) t)
          | (defaxiom fvar expr)
   expr ::= const | var
          | (cond (expr expr) ...)
          | (let ((var expr) ...) expr)
          | (fvar expr ...)
    sig ::= (fvar (var ...) t)
  const ::= t | nil | number
```

Figure A.2: A subset of ACL2 syntax.

```
  P ::= (I ... M ... C ... E)
  I ::= (interface Ivar Fun ... Thm ...)
  M ::= (module Mvar Imp ... def ... Exp ...)
  C ::= (compound Mvar = Mvar (Mvar : Ivar))
  E ::= (invoke Ivar expr)
Fun ::= (function fvar (var ...) t)
Thm ::= (theorem fvar expr)
Imp ::= (import Ivar)
Exp ::= (export Ivar (fvar as fvar) ...)
```

Figure A.3: The grammar of Modular ACL2.

barrier. Any attempt to reason about `int-fn` when verifying `map-int-fn` must be in terms of the guarantees in `INT-IFC`; there is no concrete implementation to "confuse" the theorem prover.

See figure A.3 for the full grammar of Modular ACL2, based on the ACL2 syntax from figure A.2. Modular ACL2 assumes two additional sets of variable names: *Mvar* for modules and *Ivar* for interfaces. To simplify the model, we restrict each compound module to link only two modules across a single interface, and require a program to define all interfaces first, followed by primitive modules, and then compound modules. Each program ends with an entry point expression, which may invoke functions from a specified interface provided by the module immediately preceding it.

A.2 Modeling Modular ACL2

Our model defines the meaning of Modular ACL2 in terms of ACL2. We have two separate interpretations of programs: one for logical reasoning and one for execution. The logical form of a program consists of a set of conjectures about individual modules. They represent the claim that each module satisfies its exported interfaces for any valid implementation of its imported interfaces. Programmers can use this form to verify each module of a program in isolation.

The executable form is a concrete implementation of the full program, including a fully

get-def-names	:	*(def ...)* → *(fvar ...)*	Extracts defined names.
join-imports	:	*(imp ...)* → *(imp ...)*	Removes duplicate imports.
prefix	:	*Var*, *Var* → *Var*	Adds a prefix to a variable.
prefix-names	:	*Var*, *(Var ...)* → *(Var ...)*	Prefixes multiple variables.
rename-[term]*	:	*(fvar ...)*, *(fvar ...)*, [term] → [term]	Renames identifiers in a given kind of term.

Figure A.4: Signatures of metafunctions used in the model.

linked form of the final module and an entry point expression that invokes the module. Programmers can execute this form of the program, relying on the properties of modules verified in the logical form without having to reprove them each time they are linked.

In our presentation of the PLT Redex model, some common or simple functionality is defined in separate metafunctions, summarized in figure A.4. The figure describes the general form of term(s) supplied to and produced by the metafunctions. Note that prefix and prefix-names operate on any kind of variable (written *Var*); specific applications use specific kinds of variables (such as *fvar*).

The various rename- functions perform a series of substitutions on function and theorem names, replacing all occurrences (bound and binding) of names from the first list with the corresponding name from the second list. The rules shown here use rename-var*, rename-expr*, rename-def*, and rename-export* for terms belonging to *fvar*, *expr*, *def*, and *Exp*, respectively.

We make several assumptions about variable names in Modular ACL2 programs so our translations can create new names that are both unique and readable. First, module and interface names must all be unique, as must function and theorem names in a given module. Second, wherever new names are needed, we concatenate the name of the source module or interface with the name of the definition, e.g. `PREFIX.name`. For these names to be available, the initial program must not contain such "dotted" names, and modules must only import or export a given interface once.

A.3 Logical meaning of modules

The first half of our model establishes the logical meaning of Modular ACL2 programs on a per-module basis. We translate each primitive module into a conjecture that if its imported functions satisfy the theorems in their respective interfaces, then its exported functions satisfy their properties as well. If ACL2 admits the correctness of this conjecture for abstract imports, then the module may be safely linked with concrete implementations and executed. The correctness of compound modules follows from the correctness of their

$(I \ldots M_{pre} \ldots M\ M_{post} \ldots C \ldots E) \longrightarrow (I \ldots M)$ [choose module]

$(I_{pre} \ldots I\ I_{post} \ldots$ $\longrightarrow$ $(I_{pre} \ldots I\ I_{post} \ldots$ [import interface]
(module *Mvar* (module *Mvar*
Imp ... *Imp* ...
(import *Ivar*) (defstub $fvar_{fun}$ (*var* ...) t) ...
def ... (defaxiom $fvar_{thm}$ *expr*) ...
Exp ...)) *def* ...
Exp ...))

where I = (interface *Ivar* (function $fvar_{fun}$ (*var* ...) t) ... (theorem $fvar_{thm}$ *expr*) ...)

$(I_{pre} \ldots I\ I_{post} \ldots$ $\longrightarrow$ $(I_{pre} \ldots I\ I_{post} \ldots$ [export interface]
(module *Mvar* (module *Mvar*
def ... *def* ...
(export *Ivar* ($fvar_{int}$ as $fvar_{ext}$) ...) (defthm prefix⟦*Ivar*, *fvar*⟧
Exp ...)) rename-expr*⟦($fvar_{ext}$ ...), ($fvar_{int}$ ...), *expr*⟧)
...
Exp ...))

where I = (interface *Ivar Fun* ... (theorem *fvar expr*) ...)

$(I \ldots (\texttt{module}\ Mvar\ def \ldots)) \longrightarrow (def \ldots)$ [finish]

Figure A.5: Rules for computing the logical meaning of modules.

primitive constituents.

We define the logical meaning of each module in three parts: a set of assumptions about its imports, a copy of its definitions, and a set of guarantees about its exports. We construct these parts by a translation from the original module into ACL2. The translation is formulated as a reduction relation with intermediate steps so that we can inspect the process.

Figure A.5 shows the reduction relation. The reduction rules choose a module, gradually replace its interface imports and exports with definitions, and extract the definitions at the end. The first rule chooses a primitive module nondeterministically and drops the remaining modules and the program entry point.

The second rule replaces a single import in a module with the contents of the imported interface, converted to stubs and axioms. This gives the module a self-contained, abstract definition of the imports. The new definitions use their original names, which were already reserved in the module's namespace by the import declaration.

The third rule replaces an export declaration in a module with the theorems of the exported interface as conjectures. The rule updates their bodies to refer to the module's implementation of the interface's functions, and assigns the theorems names in the module's namespace. These names are provably unique because of our name conventions, as described above.

The fourth rule completes the translation to ACL2 once a module contains no import or export declarations by producing the definitions inside the module. Note that the resulting ACL2 program may introduce unverified or unsound axioms from imported interfaces. Each

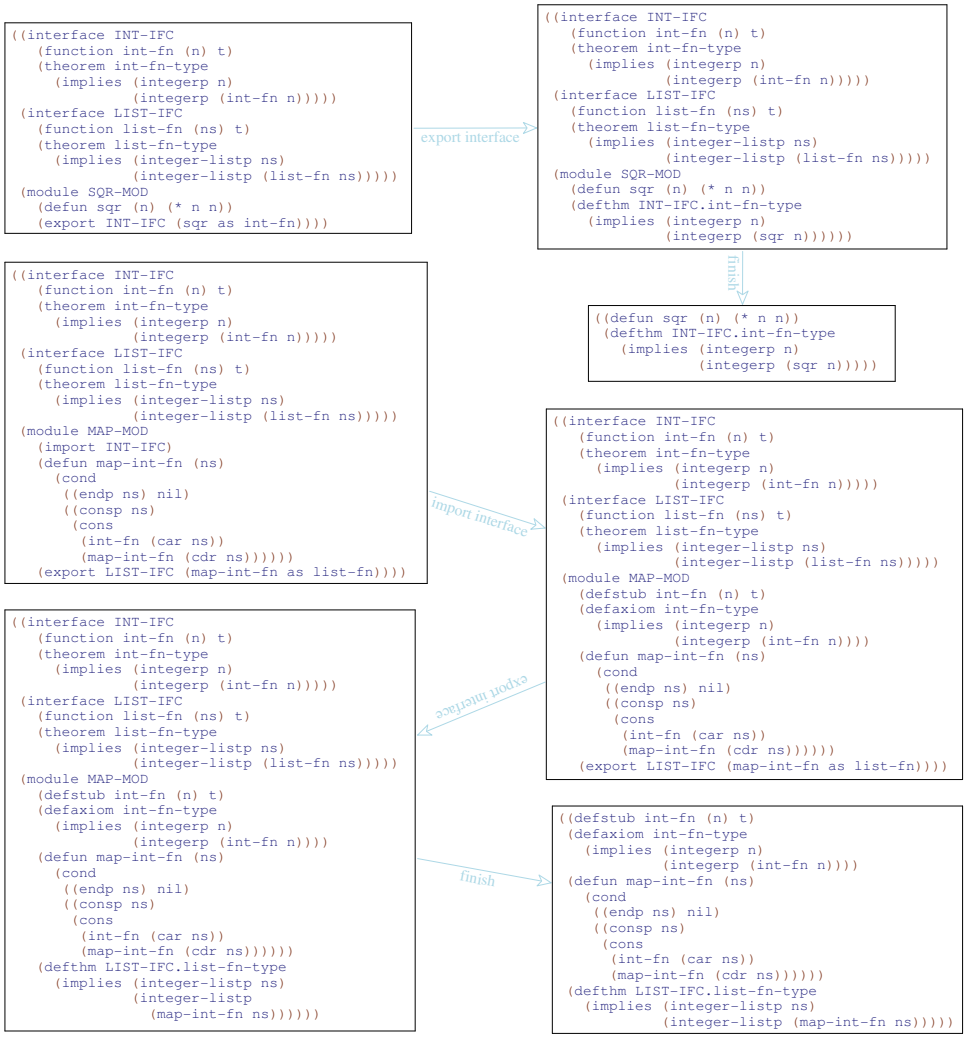

Figure A.6: Transforming SQR-ALL to its logical form.

module should be verified in a separate ACL2 session so that no other proofs may rely on these assumptions.

Figure A.6 shows steps in the transformation from SQR-ALL's modules to their logical meaning in ACL2. We omit the first step in which the module nondeterministically chooses either `SQR-MOD` or `MAP-MOD`; the two reduction paths begin with the program's interfaces followed by the respective chosen module. The top three boxes depict the transformation of `SQR-MOD`; the bottom four correspond to MAP-MOD.

The first step for `SQR-MOD` replaces the exported interface `INT-IFC` with its theorems. The "export interface" rule inserts `int-fn-type` as a conjecture with a new name and replaces the reference to the exported function `int-fn` with its implementation, `sqr`. In the second step there are no more interfaces to import or export; the "finish" rule produces

$$
\begin{array}{l}
(I_{pre} \ldots\ I\ I_{post} \ldots \\
\quad M_{pre} \ldots\ M_{one} \\
\quad M_{mid} \ldots\ M_{two} \\
\quad M_{post} \ldots \\
\quad (\texttt{compound}\ Mvar_{name} = \\
\qquad Mvar_{imp}\ (Mvar_{exp} : Ivar)) \\
\quad C_{post} \ldots \\
\quad E_{entry})
\end{array}
\longrightarrow
\begin{array}{l}
(I_{pre} \ldots\ I\ I_{post} \ldots \\
\quad M_{pre} \ldots\ M_{one}\ M_{mid} \ldots\ M_{two}\ M_{post} \ldots \\
\quad (\texttt{module}\ Mvar_{name} \\
\qquad \text{join-imports}[\![(Imp_{pre} \ldots\ Imp_{post} \ldots\ Imp_{exp} \ldots)]\!] \\
\qquad \text{rename-def*}[\![old\text{-}fvars_{exp}, new\text{-}fvars_{exp}, def_{exp}]\!] \\
\qquad \ldots \\
\qquad \text{rename-def*}[\![old\text{-}fvars_{imp}, new\text{-}fvars_{imp}, \\
\qquad\qquad \text{rename-def*}[\![(fvar_{fun} \ldots), new\text{-}fvars_{fun}, \\
\qquad\qquad\qquad def_{imp}]\!]\ ]\!] \\
\qquad \ldots \\
\qquad \text{rename-export*}[\![old\text{-}fvars_{imp}, new\text{-}fvars_{imp}, \\
\qquad\qquad \text{rename-export*}[\![(fvar_{fun} \ldots), new\text{-}fvars_{fun}, \\
\qquad\qquad\qquad Exp_{imp}]\!]\ ]\!] \\
\qquad \ldots) \\
\quad C_{post} \ldots \\
\quad E_{entry})
\end{array}
\qquad \text{[link (export first)]}
$$

where $I = (\texttt{interface}\ Ivar\ Fun \ldots\ Thm \ldots)$,

$M_{one} = (\texttt{module}\ Mvar_{exp}$,
$\quad Imp_{exp} \ldots\ def_{exp} \ldots$
$\quad Exp \ldots\ (\texttt{export}\ Ivar\ (fvar_{impl}\ \texttt{as}\ fvar_{fun}) \ldots)\ Exp \ldots)$

$M_{two} = (\texttt{module}\ Mvar_{imp}$, $old\text{-}fvars_{exp} = \text{get-def-names}[\![def_{exp}, \ldots]\!]$,
$\quad Imp_{pre} \ldots\ (\texttt{import}\ Ivar)\ Imp_{post} \ldots$
$\quad def_{imp} \ldots\ Exp_{imp} \ldots)$

$old\text{-}fvars_{imp} = \text{get-def-names}[\![def_{imp}, \ldots]\!]$,
$new\text{-}fvars_{exp} = \text{prefix-names}[\![Mvar_{exp}, old\text{-}fvars_{exp}]\!]$,
$new\text{-}fvars_{imp} = \text{prefix-names}[\![Mvar_{imp}, old\text{-}fvars_{imp}]\!]$,
$new\text{-}fvars_{fun} = (\text{rename-var*}[\![old\text{-}fvars_{exp}, new\text{-}fvars_{exp}, fvar_{impl}]\!]\ \ldots)$

Figure A.7: A rule for computing the executable semantics of a program.

the definitions inside the module.

The `MAP-MOD` module has both an imported interface and an exported interface. In its first step, the "import interface" rule replaces the import of `INT-IFC` with a stub for `int-fn` and an axiom for `int-fn-type`, representing the minimum logical properties guaranteed by any implementation of `INT-IFC`. The "export interface" rule takes effect in the second step, this time filling in a conjecture about `map-int-fn` based on `list-fn-type` from `LIST-IFC`. The "finish" rule produces the final list of definitions.

The model does not construct a logical form for `SQR-ALL-MOD`. As section A.4 shows, its correctness follows from the verification of `SQR-MOD` and `MAP-MOD` and its logical form is redundant.

A.4 Executable semantics of programs

The second half of our model defines the executable semantics of Modular ACL2 programs by translation to an executable ACL2 program. We iteratively link each compound module, constructing a new, equivalent primitive module from its primitive constituents. When the last module has been linked, we link the program entry point to the module's concrete functions. Those functions and the entry point become the ACL2 program. We show a representative reduction rule in figure A.7; the three others are variations of the same

process.

The reduction rule in the figure links a single compound module, producing an equivalent primitive module. It matches the interface for linking, the exporting module, the importing module, and the compound module.

The rule constructs the resulting primitive module by concatenation and substitution of the parts of the constituent modules. It extracts the internally defined names of each module with get-def-names, constructs unique names for them with prefix-names for use in the new module, and uses rename-var* to construct the updated names for functions implementing the given interface. The new primitive module contains the union of the original modules' unresolved imports, the internal definitions of each original module, and the exports of the importing module, all with the new function and theorem names substituted for the old.

The full reduction relation for the executable semantics consists of three other rules. Two are permutations of the first. The last rule links the program entry point to the final module; the process is similar to the rule shown here.

Figure A.8 shows the transformation from SQR-ALL to an executable ACL2 program. This takes place in two steps. First, the model produces an implementation for the compound module `SQR-ALL-MOD` by linking `MAP-MOD` to `SQR-MOD` across the interface `INT-IFC`. The result includes the internal functions of both constituent modules, each given a new, unique name. The reference to `int-fn` in `map-int-fn` is replaced with a reference to its concrete implementation, `SQR-MOD.sqr`, in `MAP-MOD.map-int-fn`. Like `MAP-MOD`, the new module exports `LIST-IFC`.

The second step links `SQR-ALL-MOD`, the last module in the program, to the program's entry point. The final ACL2 program contains the function definitions from `SQR-ALL-MOD` followed by the converted entry point, which refers directly to `MAP-MOD.map-int-fn`.

Note that the middle form of SQR-ALL contains `SQR-ALL-MOD` as a primitive module that can be translated to a logical form and verified. The resulting program, shown in figure A.9, is nearly identical to the logical form of `SQR-MOD`. Both conjecture that (a form of) `map-int-fn` from `MAP-MOD` satisfies `list-fn-type` from `LIST-IFC`. The only difference is that `SQR-MOD` refers to the stub `int-fn` where `SQR-ALL-MOD` refers to `SQR-MOD.sqr`, both of which satisfy `int-fn-type` from `INT-IFC`. The correctness of `SQR-ALL-MOD` follows straightforwardly from the more general result verified for `SQR-MOD`. Thus it is not necessary to verify the compound module separately.

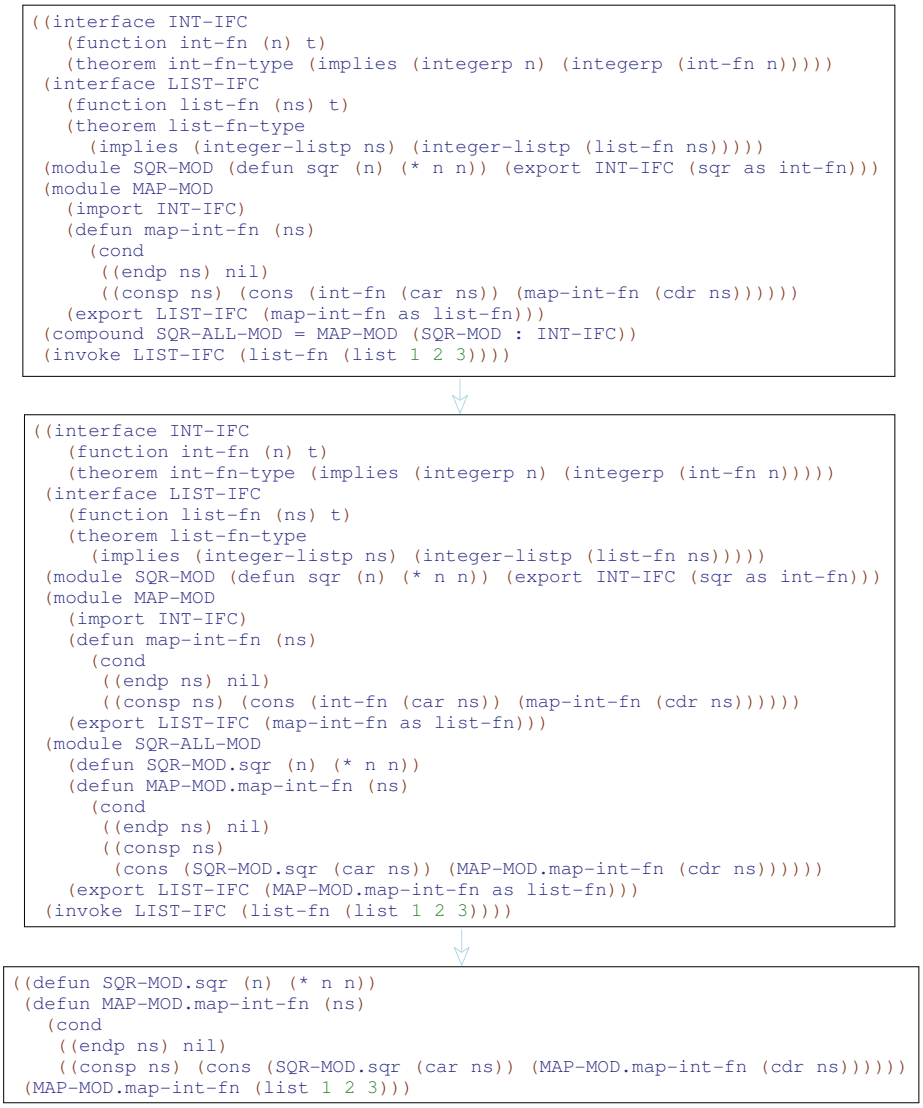

Figure A.8: Transforming SQR-ALL to its executable form.

A.5 Summary and evaluation

In this chapter, we have demonstrated a PLT Redex model for Modular ACL2, a language for developing ACL2 programs and proofs in composable, reusable components. Components contain arbitrary combinations of functions and conjectures. Aside from those exposed via interfaces, elements of one module may not interfere with another. Because primitive modules are translated to verifiable ACL2 conjectures separately, a proof attempt for one module will not interfere with another; since compound modules are linked for execution using unique renaming of internal definitions, internal names from one module will not clash with another.

```
((defun SQR-MOD.sqr (n) (* n n))
 (defun MAP-MOD.map-int-fn (ns)
   (cond
    ((endp ns) nil)
    ((consp ns) (cons (SQR-MOD.sqr (car ns)) (MAP-MOD.map-int-fn (cdr ns))))))
 (defthm LIST-IFC.list-fn-type
   (implies (integer-listp ns) (integer-listp (MAP-MOD.map-int-fn ns)))))
```

Figure A.9: The logical form of linked compound module SQR-ALL-MOD.

The language presented here includes a number of simplifications for the sake of the model which can be removed in a full implementation. For instance, ACL2 includes numerous elements not included in our grammar, including hints and other annotations for the theorem prover, packages, books, macros, and so on. Some of these features, including most of the basic tools for programming and theorem proving, can be added with straightforward extensions. Others pose more of a problem. User-defined macros complicate the renaming step of our model; Modular ACL2 can only support them by duplicating the macro expansion process.

ACL2's existing mechanisms for separating program components, such as packages, books, encapsulation blocks, and local definitions, are largely subsumed by Modular ACL2's module system. For compatibility with existing ACL2 programs, we may investigate a mechanism for converting a book to a module and vice versa. We have not yet explored how modules would interact with the other mechanisms in the same program.

Programs need not be restricted in order, nor in number of entry points. Conceptually, a Modular ACL2 program can interleave interface definitions, module definitions, and top-level expressions. This restriction simplifies the model but does not alter its expressiveness; any program can be reordered to fit the current grammar.

Compound modules in our language link two other modules across a single interface. They can be expanded to link together an arbitrary number of modules. This feature is a convenience, but can be expressed in our model by decomposing any compound module into a sequence of smaller compound modules that link incrementally.

The naming conventions in our language allow our model to create readable, unique names for new definitions. A full implementation must enforce the restrictions on names or find a more permissive mechanism for combining components. An implementation that hands off ACL2 code to the standard ACL2 implementation must generate unique names that are at least readable enough that ACL2's output can be correlated back to the original source code.

For the logical meaning of Modular ACL2, only a module's exported theorems are renamed. An implementation could remove the naming restrictions by analyzing the full

module to generate unique names, or by requiring programmers to explicitly declare names for exported theorems as well.

The executable semantics of Modular ACL2 renames every definition, possibly several times. However, the executable behavior of ACL2 can be simulated in other languages such as Lisp or Scheme, as has been done in PLT Scheme with DrACuLa Vaillancourt et al. [2006]. These languages can implement modules and linking without renaming, for instance by implementing modules as closures and passing imported functions as values.

The current design of Modular ACL2 borrows many elements from PLT Scheme's unit system Flatt and Felleisen [1998b]; Owens and Flatt [2006a] and is reminiscent of early module systems Wirth [1977, 1979, 1983a]. Module implementations and specifications are separate entities. Like units, each component has a number of inputs and outputs described by named specifications. We maintain the principle of external connections Flatt [2000]: modules do not refer to an implementation for their imports; instead, components may be combined by linking any import and export that share a specification. Unlike units, our system does not allow cyclic links; this potentially introduces new recursion and interferes with ACL2 termination proofs.

Our module system does not have some of the features demonstrated by the ML module system MacQueen [1984]. We do not support a type system for Modular ACL2, nor do we have a subsumption relation on interfaces. Modular ACL2 also does not have sharing constraints or any other way to declare a relationship between two interfaces. This is an important feature which we will explore as we expand our implementation.

Separate compilation is an important consideration for a module system. The executable form of Modular ACL2 does not support separate compilation directly in ACL2, as the code changes each time it is linked. Implementations in other languages, however, can support separate compilation; for instance, PLT Scheme's unit system supports separate compilation and all of the module features of Modular ACL2's executable semantics.

The logical form supports separate verification. As shown in sections A.3 and A.4, once a module's proof obligations are discharged it may be linked safely and without further proofs. Modular ACL2 modules can be verified by ACL2 but not certified as books, because books may not contain `defaxiom`. We hope to find a logical form for modules that can be certified; for now, we verify modules at the top level, outside of any book, where axioms are permitted.

Bibliography

Jacek Chrzaszcz. Implementing modules in the Coq system. In *Proc. 16th International Conference on Theorem Proving in Higher Order Logics*, pages 270–286. Springer, 2003.

William Clinger and Jonathan Rees. Macros that work. In *Proceedings of the 18th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 155–162. ACM Press, 1991.

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

Ryan Culpepper, Scott Owens, and Matthew Flatt. Syntactic abstraction in component interfaces. In *Proceedings of the Fourth International Conference on Generative Programming and Component Engineering*, pages 373–388. Springer, 2005.

Derek Dreyer and Andreas Rossberg. Mixin' up the ML module system. In *Proc. 13th ACM SIGPLAN International Conference on Functional Programming*, pages 307–320. ACM Press, 2008.

R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, 1992.

Carl Eastlund and Matthias Felleisen. Making induction manifest in modular ACL2. In *Proc. 11th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, pages 105–116. ACM Press, 2009a.

Carl Eastlund and Matthias Felleisen. Toward a practical module system for ACL2. In *Proceedings of the 11th International Symposium on Practical Aspects of Declarative Languages*, pages 46–60. Springer, 2009b.

Carl Eastlund and Matthias Felleisen. Hygienic macros for ACL2. In *Trends in Functional Programming, 11th International Symposium, TFP 2010*, pages 84–101. Springer, 2010.

William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. IMPS: An interactive mathematical proof system. *Journal of Automated Reasoning*, 11(2):213–248, 1993.

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

Matthew Flatt. Composable and compilable macros: You want it *when?* In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming*, pages 72–83. ACM Press, 2002.

Matthew Flatt and Matthias Felleisen. Units: cool modules for HOT languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 236–248. ACM Press, 1998a.

Matthew Flatt and Matthias Felleisen. Units: cool modules for hot languages. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 236–248, New York, NY, USA, 1998b. ACM Press. ISBN 0-89791-987-4. doi: http://doi.acm.org/10.1145/277650.277730.

Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Inc., 2010. `http://racket-lang.org/tr1/`.

Matthew Raymond Flatt. *Programming Languages for Reusable Software Components*. PhD thesis, 2000. Adviser-Matthias Felleisen.

Timothy G. Griffin. Notational definition—a formal account. In *Proc. 3rd Annual Symposium on Logic in Computer Science*, pages 372–383. IEEE Press, 1988.

Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proceedings of the 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 123–137. ACM Press, 1994.

Florian Kammüller, Markus Wenzel, and Lawrence C. Paulson. Locales: a sectioning concept for Isabelle. In *Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics*, pages 149–166. Springer, 1999.

Matt Kaufmann and J Strother Moore. Design goals of ACL2. In *Proceedings of the 3rd International School and Symposium on Formal Techniques in Real Time and Fault Tolerant Systems*, pages 92–117. Christian-Albrechts-Universität, 1994.

Matt Kaufmann and J Strother Moore. A precise description of the ACL2 logic. Technical report, University of Texas at Austin, 1998.

Matt Kaufmann and J Strother Moore. Structured theory development for a mechanized logic. *Journal of Automated Reasoning*, 26(2):161–203, 2001.

Matt Kaufmann and J Strother Moore. *ACL2 Documentation*, 2009. `http://userweb.cs.utexas.edu/users/moore/acl2/current/acl2-doc.html`.

Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, 2000.

Casey Klein. Experience with randomized testing in programming language metatheory. Master's thesis, University of Chicago, 2009.

Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 151–161. ACM Press, 1986.

Xavier Leroy. Manifest types, modules, and separate compilation. In *Proceedings of the 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 109–122. ACM Press, 1994.

David MacQueen. Modules for Standard ML. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 198–207, New York, NY, USA, 1984. ACM Press. ISBN 0-89791-142-3. doi: http://doi.acm.org/10.1145/800055.802036.

F. J. Martín-Mateos, 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*, pages 188–201. ACM Press, 2002.

Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

Scott Owens and Matthew Flatt. From structures and functors to modules and units. *SIGPLAN Not.*, 41(9):87–98, 2006a. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1160074.1159815.

Scott Owens and Matthew Flatt. From structures and functors to modules and units. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming*, pages 87–98. ACM Press, 2006b.

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

Donald Sannella. Formal program development in Extended ML for the working programmer. In *Proceedings of the 3rd BCS/FACS Workshop on Refinement*, pages 99–130. Springer, 1991.

The Coq Development Team. *The Coq Proof Assistant Reference Manual*, 2006. `http://coq.inria.fr/V8.1pl3/refman/index.html`.

Dale Vaillancourt, Rex Page, and Matthias Felleisen. ACL2 in DrScheme. In *ACL2 '06: Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications*, pages 107–116, New York, NY, USA, 2006. ACM Press. ISBN 0-9788493-0-2. doi: http://doi.acm.org/10.1145/1217975.1217999.

Niklaus Wirth. Modula: a language for modular multiprogramming. *Softw., Pract. Exper.*, 7(1):3–35, 1977.

Niklaus Wirth. The module: A system structuring facility in high-level programming languages. In Jeffrey M. Tobias, editor, *Language Design and Programming Methodology*, volume 79 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 1979. ISBN 3-540-09745-7.

Niklaus Wirth. *Programming in Modula-2*. Springer-Verlag, Berlin, Germany, second edition, 1983a. ISBN 0-387-12206-0.

Niklaus Wirth. *Programming in Modula-2*. Springer, 1983b.

All that is gold does not glitter,

Not all those who wander are lost.

— J. R. R. Tolkein, *The Lord of the Rings*