Modular Metatheory for Extensible Languages

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Dawn Michaelson

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Gopalan Nadathur, Eric Van Wyk

August, 2024

Acknowledgements

I'd like to thank my advisors, Eric Van Wyk and Gopalan Nadathur, for passing on their expertise in the areas of extensible languages and logic, respectively, as well as for sharing their time. This work would not have been possible without them.

I would also like to thank the rest of my committee members, Favonia, Paul Garrett, and Sanjai Rayadurgam, for their time, as well as their helpful comments and interesting conversations over the years.

Finally, I would like to thank my labmates over the years for interesting conversations: Travis Carlson, Nathan Ringo, Lucas Kramer, Allie Hanson, Ian Kariniemi, Luke Bessant, Mary Southern, Dan DaCosta, Duanyang Jing, Nathan Guermond, Chase Johnson, and Terrance Gray. Especial thanks to Lucas Kramer for helping me with Silver, particularly in fixing the bugs I found by using it in ways no one had before.

Work on this dissertation has been partially supported by a Doctoral Dissertation Fellowship from the University of Minnesota. The author was also supported by NSF Grant 2123987 while working on this project. Opinions, findings, and conclusions or recommendations expressed in this thesis should be understood as mine. In particular, they do not necessarily reflect the views of the National Science Foundation.

Abstract

The features available in a programming language affect how easy it is to write programs for different purposes using it. *Extensible languages* allow specifications of language features to be developed independently of one another, then composed to form a new language containing all the features from all the specifications included. This lets programmers choose language features they find useful for each program they write.

However, because the specifications of different language features are developed independently of one another, we do not know the language resulting from composition is well-formed in desired ways. In particular, we don't know what properties should be true of the language, nor do we have a proof that they are true; that is, we do not have *metatheory*, a set of properties proven to be true, for the language. Metatheoretic properties are known to hold for any program written in the language, providing useful information to programmers about what can and cannot happen in a program. For example, the type preservation property can be stated as terms with a certain type evaluate to values of the same type, and guarantees well-typed programs cannot encounter type errors during evaluation. In an extensible setting, proving properties will hold for any composed language is difficult because the full language is not created until the time of composition.

This thesis develops a modular approach to establishing the metatheory of extensible languages written using a framework where modules specifying language features may build on others. In this approach, each specification module independently introduces new metatheoretic properties expected to hold for any composed language that includes the module. The work of proving these properties for any composed language is distributed across modules, with each module contributing proofs for the new properties it introduces and those of the modules on which it builds. These proofs are written in the limited context of a single module's knowledge but need to reason about the larger contexts of composed languages, which may include constructs unknown to the module writing the proof. To address this, we utilize generic reasoning to handle the currently-unknown additions that may

be present in a composed language. When a language composition is built, the proofs from each module are then used to create a composed proof for each property. These composed proofs guarantee each property from each module included in the composition holds for the composed language, bringing the benefits of metatheory to the extensible setting.

In addition to developing a reasoning framework for modular metatheory, we implement it in an interactive proof assistant named Extensibella. Extensibella aids users in writing modular proofs. It does this by adding an extra layer to Abella, an existing proof assistant. This extra layer checks the proofs are valid in the context of a single module and that generic reasoning is carried out correctly, then passes proofs to Abella to check their validity in the logic $\mathcal{G}$ in which it constructs proofs. Extensibella also includes functionality to compose the proofs from individual modules to form full proofs of properties for any composed language. To support this implementation, we have also implemented the framework for language specification about which we reason in a system named Sterling. Sterling allows users to write language specifications and check them for their modular validity, then produces specifications Extensibella can read and about which it can reason.

Finally, using these implementations, we have developed a set of example extensible languages and their metatheory. These applications suggest our framework's structure does not induce significant limits on what we can prove, so modules can introduce interesting new syntax, semantics, and metatheoretic properties. They also allow us to examine the trade offs in our framework between the freedom given to modules to introduce interesting new syntax and semantics and the ability to introduce and prove new metatheoretic properties.

Contents

List of Tables

List of Figures

List of Definitions and Theorems

Chapter 1

Introduction

Different programming language features are useful in writing programs for different tasks. For example, some problems are easier to solve using imperative languages, while others lend themselves well to functional languages. Solutions to some problems benefit from domain-specific language features targeting niche areas, features that often aren't included in general-purpose languages. This leads to using different languages for different tasks, but sometimes programmers need diverse sets of features for the same task, features not commonly found together in the same language. One solution is to use *extensible languages*. Extensible languages allow composing specifications of language features, written independently of one another, from a library of language modules to form a single programming language that contains the features from all the included modules. This allows programmers to pick and choose the language features they want to use just as they might pick and choose the libraries they want to use.

This general approach to language development has been used in a number of previous applications. SugarJ [9] lets Java programming libraries introduce new syntax along with new classes and functions, such as syntax for pairs as part of a package providing a class implementing pairs, as well as compile-time checking of correctness specific to the new features being introduced. The JastAdd Extensible Java Compiler [8] also extends Java, allowing new feature specifications that include both new static checks and new syntax, such as a non-null annotations and a checker to ensure variables marked as non-null will never be null. AbleC [17] extends the C programming language, similarly allowing feature specifications to add new static checks and syntax to the C language. Features added in

this way include algebraic datatypes and pattern matching on their constructors and syntax for regular expressions, along with a static check they are valid regular expressions.

The difficulty with extensible languages such as these is in knowing that languages built from independent specifications are well-formed in relevant ways. Because the specifications of the features are developed independently, there is no one to check that all possible combinations of features work together correctly. This is in contrast to non-extensible languages, where one set of maintainers develops the full language and ensures all parts work correctly together.

Perhaps the most important part of a language's well-formedness, and the one in which we are primarily interested in this thesis, is its *metatheory*. Metatheory is the set of properties proven about a language, and thus known to be true for all programs written in it. A common metatheoretic property is type preservation, that well-typed expressions evaluate to values of the same type (*e.g.*, an integer-typed expression evaluates to an integer value), guaranteeing well-typed programs do not encounter type errors. When a language is built by composing independent specification modules introducing features, proving something about the result of composing any set of modules is difficult because what will be included in the composition is not known until the composition is made. However, we cannot wait to prove properties until a composition is built as the developers of the modules, who know their parts of the language best, are no longer involved. We also cannot prove the properties we want for all possible compositions, as we do not assume all modules are known at any point; a new one could be written at any time.

In this thesis, we present a solution for modularly developing the metatheory of extensible languages. Our approach permits any module to introduce new metatheoretic properties that will hold for any composed language that includes the module. The work of proving these properties are true is distributed across modules, with a guarantee a full proof of each property then exists for any composed language, guaranteeing all properties from all modules included in a language composition are true for the composed language. The distribution of both the introduction and proving of properties creates a truly modular approach to metatheory for extensible languages.

1.1 A Vision for Language Extensibility

Let us expand on what we mean by extensible languages and how we envision them being used. As alluded to above, an extensible language has a library of modules introducing language features, where sets of modules may be composed to form different languages with diverse sets of features. While modules may be completely independent of each other, they may alternatively build on each other, adding to the definitions given by others. A common structure is to have a *host language*, a module that forms a base on which other modules build, and *extensions*, modules that extend the features given by the host language. Each module, whether a host language or an extension, should be able to introduce new syntax and extend existing semantic analyses to that new syntax, as well as introduce new semantic analyses and define them for the language.

The best way to understand our vision for extensible languages is through an example. Suppose we have an imperative host language with basic constructs in it. This language defines static typing as well as evaluation, along with proving type preservation. One programmer wants to use this language, but he also wants matrices and matrix operations it does not include. The language library may already contain a module introducing such features, or he or a friend can write one. This module adds new constructs for creating matrices and working with them, and it extends the definition of typing to check matrices used in the same operation contain the same type (*e.g.*, only matrices whose elements are integers are added to other matrices whose elements are integers) and the definition of evaluation to carry out the matrix operations. The module author also extends the proof of type preservation to include the new matrix operations. Our programmer can now write his program using a language composing both the host language module and the matrix module, using the matrix features in the same way he uses any other feature in the language.

Another programmer also wants to use our extensible language, but she needs to handle sensitive data in her program, and she wants the language to check she doesn't accidentally leak this data to places that shouldn't have access to it. This can be accomplished by labeling variables that hold sensitive data, then having a check that the sensitive data does

not leak. Such an analysis has been developed [32] and was implemented in the JFlow [31] modification of Java. As with matrices, a module adding this check as a language feature can be written. Unlike matrices, this is adding an entirely new semantic analysis to the language. The matrix module extends the existing semantics by adding new cases to evaluation and typing specific to matrices, but this information security check is separate from the existing semantics of the language. The module writer also proves a new metatheoretic property showing this analysis guarantees no information leaks can occur in programs that pass the security check. Our programmer can use a language composition including both the host language module and the security module to get a full language including typing, evaluation, and information flow security checking with a guarantee that her program does not leak sensitive information.

Consider a third programmer. He has a programming task where he needs to use matrices but he also needs to handle sensitive data. He can use a composed language including *both* the matrix and security modules simply by declaring he wants to use both, with tools automatically creating a compiler or interpreter with matrices and the security check; no extra work is needed to build the composition, even though the security and matrix modules were written independently of one another. In this composed language, the security check is defined for the matrix operations as well as the host language's constructs, and it checks the whole program for leaks of secure information, in addition to typechecking ensuring the program, including matrix operations, is well-typed. Moreover, the metatheoretic property about the security check still holds, and the check still guarantees sensitive information does not leak even though the matrix module that was not known when the security analysis was written is included in the composed language. Ensuring properties such as this one for any language composition, especially ones with independent modules like the matrix and security modules, is the focus of this thesis.

The compositionality we see in this third scenario is the main benefit of extensible languages relative to other approaches to language construction. While the JFlow implementation guarantees sensitive information is not leaked, it does not give a way to combine this with other modifications of Java, such as Coffer [33] that adds resource management

facilities for ensuring protocols are followed, and thus using both facilities together requires re-implementing both sets of features in a new, shared implementation.

Note that, in our vision, not only do we have such compositionality of modules, but each *programmer* is choosing the modules he or she wants to be included in a composed language; the compositionality of language features does not depend on a language expert to put a composition together or test it for correctness. The only part where language experts may be needed is in writing the modules, not using them. Then anyone can use a language with the features they want, and with the appropriate metatheoretic guarantees for those features, as long as each feature independently exists.

1.2 Requirements for Complete Modularity

The vision we have laid out for extensible languages requires *complete modularity* of the language. We define this as the language library being *open* and composition of language modules being *automatic*, both of which we discuss below. These requirements can be applied to all parts of a language's specification, its concrete syntax, its abstract syntax and semantic definitions, and its metatheory. Our focus in this thesis will be on metatheory, with semantics as a secondary focus necessary for considering metatheory. This is different from most existing extensibility frameworks that only consider concrete syntax or semantics, not metatheory. Most existing frameworks also do not follow both our modularity requirements for the parts of language specifications they do consider, generally being either open or automatic, but not both, rendering them insufficient for realizing our vision from the previous section.

Our first requirement for complete modularity is that the language library should be open, allowing anyone to contribute a new language module at any time. We saw this in the previous section, where the matrix and information flow security extensions could be written and added to the language library if they were not already present. A language cannot truly be considered extensible if new modules cannot be added; in that case it is simply a way for organizing a language's features. Some prior approaches to making

extensions or modifications to languages, such as Polyglot [33] for making modified versions of Java, have been closed, not allowing new additions to existing languages. Qualitatively, in addition to the library being open, modules should also be able to make meaningful additions to a language. What are "meaningful" additions cannot be quantified, of course, but we expect an extensible language to allow module writers some freedom in defining language semantics. A system of macros that simply expand to set code would not fit our expectations.

The second requirement for complete modularity is that composition of language modules should be automatic. Any subset of the modules in the library can be used to create a composition, modulo a requirement by one module that another be included in the set (*e.g.*, the security and matrix modules from our example both require the host language module to be included in a composition with them, as they build on its definitions). This requirement ensures a language expert is not needed to build a composition, allowing programmers to create their own languages as we saw with the third programmer choosing both extensions in the previous section. Many prior frameworks for extensibility require *glue code*, modifications and additions that make the specifications from different modules work together, to form a composition [5, 6, 13, 22]. The necessity of changing or adding definitions to make a composition work prevents average programmers from building language compositions, as they are not experts in languages and cannot make the necessary changes. Consider especially the JastAdd [13] system for writing extensible languages. JastAdd tries to accomplish much of what we laid out in our vision: modules may be freely added to a language library, and they may add new abstract syntax (*e.g.*, the matrix module) and new semantics (*e.g.*, the security module). However, JastAdd requires glue code to form a composition in the general case. For example, the definition of the security analysis for matrix operations would need to be written for the composition, preventing an average programmer from creating the composition.

The focus of complete modularity is on allowing programmers who are not experts in programming languages to make languages with the features they want, modulo those features being available in a language library, and on allowing anyone with language expertise

to add a new feature module to the language library. This democratizes language construction, both for language users and language writers. However, this is not the focus of most existing frameworks, which is why they do not follow our modularity requirements. If new modules cannot be added, all compositions are known, and that the language is well-behaved for all compositions can be checked by checking each possible combination. If composition is not automatic, a language expert can ensure its specification is sensible, and can directly specify any new pieces needed in the composition, such as the definition of the security check for matrices. Such systems allow reuse of existing components, reducing the work required to build a particular composed language, but not necessarily eliminating it.

There are a few extensibility frameworks that fit our view of complete modularity for some aspects of language specification, but cannot fulfill our vision either due to not allowing extensions to introduce new semantic analyses, such as the aforementioned information security check, or due to not having a framework for modular metatheory. Modular Structural Operational Semantics (MSOS) [29] allows modules to extend and to modify existing semantic analyses, such as relations defining evaluation, but does not allow introducing new semantic analyses. Thus MSOS is completely modular in extending the existing language semantics, but not in adding new semantic analyses, and cannot fulfill our vision.

Similarly, a framework we call *complementary components* [5, 7, 30, 35] has modules build on a shared set of definitions and is completely modular for extending shared language semantics. Approaches exist for developing the metatheory of such languages, but tend either not to be completely modular because they require glue proofs for metatheory composition or not to be powerful enough to prove desired properties. As with MSOS, modules in complementary components also cannot introduce new semantic analyses, and thus cannot fulfill our vision even with more robust approaches to developing metatheory.

The Silver attribute grammar system [38] has a completely modular extensibility framework for specifying language semantics. In addition to extending the definitions of existing analyses, this framework supports adding new semantic analyses. In language composition, missing definitions, such as the definition of the security check for matrices and matrix operations, are created automatically. Silver is paired with the Copper parser generator [25]

that allows writing completely modular specifications of concrete syntax. However, the existing metatheory approach for Silver's extensibility framework [19] is too restrictive, falling short of the qualitative requirement of modules being able to make meaningful additions. Thus we see our vision requires a new framework that will be completely modular for both semantics and metatheory.

1.3 Our Framework for Extensibility

We propose an extensibility framework for language definitions and a reasoning framework for proving metatheory of languages specified using the extensibility framework. Both fit our definition of complete modularity, with an open library of modules, as well as automatic composition of language semantics for the language extensibility framework and automatic composition of sets of properties and their proofs for language metatheory for the reasoning framework. We consider each in turn.

1.3.1 Defining Languages

Our extensibility framework for defining language semantics borrows heavily from the one introduced by Silver [38] mentioned in the previous section, with our changes making it more general and more useful for reasoning. Individual language modules may introduce new syntax categories and constructors of them, as well as new semantic relations and define them. Modules may also build on other modules, extending their semantics. Extension modules may add new constructors of existing categories and may define existing semantic relations for them. An example of this would be the matrix module above defining matrices and matrix operations as constructors of an existing syntax category for expressions, then defining the existing typing and evaluation relations for them. However, our framework imposes restrictions on extending the definitions of existing semantic relations in extension modules. The purpose of this is to maintain exactly the same meanings of existing relations for existing constructs. This ensures all modules building on an existing one can understand its constructs in any composition.

Composition of modules is the union of all syntax categories, constructors, semantic relations, and rules defining them. Note that this union necessarily may contain holes, however, as we can have independent modules introduce new constructors for and new relations over the same syntax category, as in the security and matrix modules. In a composition, the union of existing definitions will not provide a definition of the security analysis for matrix operations that we need for using them together in the same program. Whereas many existing frameworks use glue code to solve this problem, our need for automatic composition does not allow this. Instead, each relation introduced by an extension module gives a default rule for itself, which is then used to define it for new constructs from other extensions. This is how the security analysis is defined for matrix operations, with the security module giving a default rule and the composition process instantiating the default rule for each new constructor introduced by the matrix module. Thus we have automatic composition while also having complete definitions for new semantic analyses in composition.

1.3.2 Modular Metatheory

Our reasoning framework builds on our extensibility one, assuming the language semantics are given by modules in such a language library. As each module can introduce syntax categories and semantic relations, so also can they introduce new properties. For example, the host language from the example above can introduce the type preservation property and the security extension module can introduce properties showing its analysis guarantees a program cannot leak sensitive information. Composition of properties is the union of the properties introduced by all the modules included in a language, so a composition including the security extension and the host language would be expected to have both the type preservation and security properties.

Showing a composed language has the expected properties requires proofs. Our proofs will be derivations in a particular logic. As with all other elements of language specifications, property proofs will be introduced by modules, then composed to create proofs for the full language. Just as the full definition of a semantic relation in our extensibility framework is given by individual modules introducing independent rules, so the proofs of metatheoretic

properties are given by individual modules writing independent pieces of them. The proof of a property is broken down into cases that correspond to the rules defining a relation, with the module introducing either the property or the rule proving each case.

As with language semantic definitions, however, this breakdown necessarily leaves holes. Some rules are introduced in the composition by instantiating default rules, as with the security analysis's default rule being instantiated for matrices. The key to automatic composition of proofs will be to handle such cases *generically*. The module introducing a property knows such cases may exist, and it knows roughly how they will be defined, as it knows the default rule that will be instantiated for any such cases. For example, the security module knows what the default rule for the security analysis is, so it can show other constructs will be secure as well in a general way, even though it does not know them specifically.

In addition to ensuring each rule in the composition has a proof, we also need to know these proofs written by individual modules will be valid in a composition. In particular, reasoning by case analysis poses a problem, as it relies on a global closed-world assumption, that all possible applicable rules, and therefore all possible cases, are known. This is not so in an extensible setting, as other modules can introduce new rules. However, we can have a *local* closed-world assumption, based on the limitations on extending the definitions of existing semantic analyses that are part of our extensibility framework. Limiting case analysis to situations where we know other modules cannot add applicable rules ensures the set of applicable rules in a composed setting is the same as in the setting of the individual module writing the proof, and thus each case arising from the case analysis in any composed setting is handled by the proof written by the module.

Proof composition is necessarily more complex than the union of sets used for other elements, as it needs to create a structure that is valid by the rules of the logic rather than simply a new set. We will require the basic structure of the modular proofs and the composed proof to be the same, so all the composition needs is to fill in the proof for each case. This can be accomplished by taking the proof cases written by the modules and, for each case for a new construct from a different module, generating a proof from the generic one. Our requirements of proofs written by modules will ensure proof composition

succeeds, creating a valid proof in the context of the composed language for each property from each included module. Thus, having a proof for any composed language for each expected property, we know all properties hold for any composed language.

1.4 Contributions

In this thesis, we first create an extensibility framework for writing extensible languages based on that used by Silver [38], but somewhat more general. In particular, whereas Silver's model is specific to attribute grammars, our reformulation applies to inference-rule-based specifications. Furthermore, ours gives more freedom in declaring how the definitions of new semantic attributes are to be extended to constructs from other modules, such as how the definition of an information flow security analysis from one extension would be defined for matrix operations from another.

The main body of this work is the formulation of a framework for understanding metatheory in the setting of extensible languages that use this extensibility framework. Metatheoretic properties may be introduced modularly, in the same manner as language semantics are introduced. These properties may then be proven modularly as well, with proofs distributed similarly to how the definitions of language semantics are distributed. The modular proofs written by the designers of individual language modules can then be used to build proofs of each property for any composed language, guaranteeing each property holds for any composed language. Going beyond individual properties, our framework ensures full sets of properties hold for composed languages, guaranteeing full sets of proofs do not have circular dependencies between properties used as lemmas.

We have implementations of both the extensibility framework in our new Sterling system and the reasoning framework in our proof assistant Extensibella. These work together, with extensible languages being defined and checked for well-formedness in Sterling, and proofs of their properties being written and checked for modular validity in Extensibella. Beyond facilitating modular reasoning, Extensibella also includes the capability to take the modular proofs and combine them to create machine-checkable proofs for any composed language,

implementing the proof composition that guarantees the modular work is valid.

Finally, we have tested our framework by developing a set of applications written in Sterling and Extensibella demonstrating the language extensibility framework is useful for writing languages and the reasoning framework is practical for proving their properties. These applications also demonstrate the trade-offs in our modular reasoning framework between the freedom of extension modules in defining the semantics of new constructs and the ability of extension modules to introduce and prove interesting properties.

1.5 Overview

Chapter 2 presents our language extensibility framework and compares it with the existing Silver framework on which we have based ours. It also introduces an extensible language with a host language and three extensions building on it that we will use as a running example in future chapters.

The next three chapters deal with the foundations of modular metatheory. First Chapter 3 presents the logic with which we will prove metatheoretic properties of languages and how we encode our languages into the logic. Chapter 4 then uses this logic in presenting our reasoning framework. We present the requirements for modules in proving both new properties they introduce and any imported from other modules that they extend. We also present how these modular proofs may be used to create a full proof of an individual property for any composed language, demonstrating the reasoning framework is sound for proving properties modularly. Chapter 5 finishes our presentation of the reasoning framework by discussing proving sets of properties. The main part of this is how we order properties for use as lemmas in the context of individual modules to ensure that we do not have circularities in sets of proofs for composed languages. This gives a guarantee that the full set of properties will hold for any composed language, building on the proofs that individual properties will hold for any composed language.

The next two chapters focus on the practical aspects of using the reasoning framework. We present our implementations of the extensibility and reasoning frameworks in Chapter 6,

discussing their uses and how we implement proof composition. Chapter 7 then discusses some examples we have developed using our tools, the lessons we have learned from them, and the trade-offs to be considered for designing extensible languages and their properties in light of extensibility, especially in light of modular proofs and generic reasoning.

Finally, we close by considering the past and the future. Chapter 8 presents prior work on modular reasoning for extensible languages, comparing them to our own reasoning framework. Chapter 9 summarizes our work and discusses possible future directions.

Chapter 2

A Framework for Language Extensibility

Language specifications define a language's syntax and semantics. Syntax descriptions identify a language's syntax categories, such as a category for expressions, and constructors for these categories. Having identified the syntax, a language specification may then define relations over it, often including both static analyses, such as those determining if a term is well-formed, and dynamic analyses defining notions related to evaluation.

Our extensibility framework organizes these definitions into modules. Each module may introduce syntax categories, syntax constructors, relations, and rules defining relations, with a full language being the composition of the pieces introduced by a set of modules. Modules are further organized into language libraries of modules that may be composed to form full languages from some subset of the modules in a library. Language libraries start with a single module, or a small set of modules, often defining a host language. Developers then write new extension modules building on those already in the library, contributing them when they are complete. The independent nature of writing extension modules leaves the language library open to extension anytime, by anyone, as the developer of a new extension does not need to know the contents of the entire library in writing a new module, only those parts on which it depends.

Language composition creates a complete language from a set of specification modules, requiring no work from the programmer creating a composition beyond specifying which modules to include. The composed language includes all the syntax categories, syntax constructors, relations, and rules from all the modules included in the composition. Additionally, it includes rules defining relations introduced by one extension for new constructors

introduced by another, these rules being created automatically for the composition to complete the definitions of relations. A composed language thus includes all features from all modules included in the composition.

The framework we present is based on that used by the Silver attribute grammar system [38] and the modular well-definedness analysis [18] it uses to check modules are well-formed, ensuring any composition will be well-formed. We distill the portions of these existing works that are generally applicable, dropping the portions specific to attribute grammars.

In the rest of this chapter, we discuss our framework for writing extensible languages. We start in Section 2.1 with how we define individual modules, then discuss in Section 2.2 the requirements for language modules individually, and language libraries in total, to be well-formed. We also define, in Section 2.3, how we compose a set of extensible language modules to form a complete language. Finally, Section 2.4 discusses specifically how our extensibility framework relates to that used by Silver.

2.1 Defining Language Modules

A module is, informally, a set of syntactic and semantic definitions that may add to the syntax and semantics already defined by other modules. Formally, a module is an 8-tuple, written as $\langle \mathbb{B}, \mathscr{C}, \mathbb{C}, \mathscr{R}, \mathbb{R}, \mathscr{T}, \mathbb{T}, \mathbb{S} \rangle$. The first item in this tuple is for specifying relationships between different modules, the next two for defining syntax, and the remainder for introducing and defining relations. Note we will write these with a superscript to identify the module to which a tuple element belongs. For example, $\mathscr{C}^M$ is the second element of the tuple defining the M module. We will look at each portion in turn.

The first element is the set of modules $\mathbb{B}$ on which a module builds; this defines the relationships between different modules in a language library. A module is aware of all the declarations in the modules on which it builds, both syntax declarations and semantic declarations, and may add to them. Modules may build on one another freely, other than a restriction that a module cannot build on itself; we require acyclicity of the builds-on

relationships of a language library. We say two modules are *unrelated* if neither builds on the other. We will sometimes refer to the inclusion of a module in $\mathbb{B}^M$ as M *importing* that module or some element that it introduces. This emphasizes the similarity between building on an existing set of modules to form a new language module and using libraries, essentially programs, in writing a new program in standard programming languages such as Java or Python. We also say a module *knows* the modules in its builds-on set, and the syntax and semantics they introduce, to contrast with *unknown* modules and their syntax and semantics, those not part of the builds-on set of a module.

Some extensible language systems limit their module structure to a host language and an open set of extensions to it. This basic structure is subsumed by our structure of modules building freely on one another, but we will often borrow the terminology of an *extension* to refer to a module building on other modules, particularly in situations where we wish to emphasize a module is adding to an existing set of definitions, or *extending* the definitions, given by a module on which it builds.

2.1.1 Specifying Syntax

This work is focused on the semantics of extensible languages, so we are interested only in the abstract syntax of modules, not concrete syntax. Thus our module tuples include only descriptions of abstract syntax, not concrete syntax. Prior work has addressed developing concrete syntax for extensible languages in a comparably extensible manner [36].

In our modules, $\mathscr{C}$ is a set of syntax categories and $\mathbb{C}$ is a set of constructors. Each constructor takes arguments of specific syntax categories and builds a term in a specific syntax category. When a constructor is introduced, its argument syntax categories and the one it builds must be known categories. The known syntax categories for a module M are those introduced in $\mathscr{C}^N$ for a module $N \in \mathbb{B}^M \cup \{M\}$, that is, the known syntax categories are those from M or one of the modules on which it builds. Some syntax categories will be extensible and others not, with a category's extensibility determined by the $\mathscr{T}$ element. We will discuss how this is specified below. What is important to the introduction of syntax constructors is that introducing a constructor building an expression in an imported

category is permissible only if that category is an extensible one. New constructors for non-extensible imported categories are not allowed.

When presenting a language module, we will represent its syntax introductions similarly to datatype declarations in a functional language such as OCaml. Figure 2.1 shows the syntax of our example language, with a host language H, an extension introducing list constructs L, and an extension for information flow security S. Our host language introduces categories for statements s, expressions e, types ty, variable names n, integers i, typing contexts Γ, and evaluation contexts γ. Note we do not specify constructors for the n or i categories in our presentation for simplicity. The categories other than statements, expressions, and types are non-extensible. This language includes Boolean and integer types, with values of those types and operations over them in the expressions. It also includes statements for declaring variables and assigning values to them, conditionals, while loops, and sequencing statements, as well as a no-op statement (*skip*).

The list extension adds to the host language by introducing a list type constructor and expressions for building and deconstructing lists. It also introduces a *splitlist* statement form for simultaneous assignment of the head and tail of a list to separate variables. For example, $\mathit{splitlist}(h, t, \mathit{cons}(\mathit{true}, \mathit{nil}))$ will result in assigning *true* to h and *nil* to t. Note the extreme simplicity of our language prevents more common but more complex statement additions by the list extension such as a for-each loop, which is why we choose the *splitlist* statement as an example.

Our other extension is the information flow security extension. This extension introduces the idea that some variables may contain sensitive information that needs to be treated specially to ensure users of a program do not gain access to it. For example, the encryption keys used in a program should be considered sensitive and users should not have access to them. To provide a way to annotate programs with confidentiality levels, the extension introduces a category *sl* for security levels with constructors *public* and *private*. The *secdecl* statement constructor then allows declaring variables with one of these security levels, with variables declared using *private* treated as holding information to be kept confidential. The extension also introduces a category for security contexts Σ associating variable names with

$\mathbb{B}^H = \varnothing$

$\mathscr{C}^H = \{s, e, ty, n, i, \Gamma, \gamma\}$

$\mathbb{C}^H$:

s	::=	$skip$	e	::=	$var(n)$	ty	::=	int
	|	$decl(n, ty, e)$		|	$intlit(i)$		|	$bool$
	|	$assign(n, e)$		|	$true$			
	|	$seq(s, s)$		|	$false$	Γ	::=	$nilty$
	|	$ifte(e, s, s)$		|	$add(e, e)$		|	$consty(n, ty, \Gamma)$
	|	$while(e, s)$		|	$eq(e, e)$			
				|	$gt(e, e)$	γ	::=	$nilval$
				|	$not(e)$		|	$consval(n, e, \gamma)$

$\mathbb{B}^L = \{H\}$

$\mathscr{C}^L = \{\}$

$\mathbb{C}^L$:

e	::=	nil	ty	::=	$list(ty)$
	|	$cons(e, e)$			
	|	$null(e)$	s	::=	$splitlist(n, n, e)$
	|	$head(e)$			
	|	$tail(e)$			

$\mathbb{B}^S = \{H\}$

$\mathscr{C}^S = \{sl, \Sigma\}$

$\mathbb{C}^S$:

sl	::=	$public$	Σ	::=	$nilsec$	s	::=	$secdecl(n, ty, sl, e)$
	|	$private$		|	$conssec(n, sl, \Sigma)$			

Figure 2.1: Builds-on sets and syntax of the example host language (H), list extension (L), and security extension (S)

security levels; we will use security contexts for checking information about the contents of private variables cannot be leaked to public variables. This extension and its analysis is a simplification of prior work on security in information flow [32].[1]

2.1.2 Specifying Language Semantics

We define language semantics via relations, given by the next two elements of a module-defining tuple $\langle \mathbb{B}, \mathscr{C}, \mathbb{C}, \mathscr{R}, \mathbb{R}, \mathscr{T}, \mathbb{T}, \mathbb{S} \rangle$. The set $\mathscr{R}$ is the set of relations introduced by a module. Each relation is given an arity and syntax categories for the types of its arguments. Just as $\mathbb{C}$ defines the categories in $\mathscr{C}$ by giving constructors for building them, so $\mathbb{R}$ defines the relations in $\mathscr{R}$ by giving rules for deriving them. Also as with constructors, the rules in $\mathbb{R}^M$ may define any known relation, and may rely on any known relation as a premise for the rule, with the set of known relations being those introduced in $\mathscr{R}^N$ for a module $N \in \mathbb{B}^M \cup \{M\}$, that is, relations introduced by M or any module on which it builds.

There is a limitation on introducing rules defining imported relations, however. We want to hold the definitions of imported relations as fixed for imported constructs. This makes it possible for extensions to understand constructs introduced by the modules on which they build, as the semantics cannot be changed by other modules in a composition. As an example of why this is important, consider addition in our example language. Our host language will define a typing relation $\Gamma \vdash e : ty$ relating a typing context assigning names to variables, an expression, and the type of that expression. It defines addition to have an integer type when the two expressions being added have integer types (*i.e.*, the expected rule for typing *add*). Suppose an extension to our example language introduced complex numbers and a type *complex* and added this new rule for typing an addition of complex numbers:

$$\frac{\Gamma \vdash e_1 : \mathit{complex} \qquad \Gamma \vdash e_2 : \mathit{complex}}{\Gamma \vdash \mathit{add}(e_1, e_2) : \mathit{complex}} \text{ T-Add-Complex}$$

[1]This particular instance is a modification of information flow security as presented by Andrew Myers at OPLSS in 2019. Lecture recordings and notes can be found online at `https://www.cs.uoregon.edu/research/summerschool/summer19/topics.php#Myers`.

In a non-extensible setting, this rule is fine and does define what we would expect the typing of adding complex numbers to be. However, the inclusion of this rule would mean an addition could have a complex number type, not just an integer type as it was defined to have in the module introducing it. Then other extensions could not rely on addition having an integer type in defining their own semantic relations for addition. Allowing new rules of this sort would make defining new semantic relations coherently relative to imported semantics difficult, as the imported semantics could be changed in a composition.

To ban extensions from introducing rules changing the semantics of established constructs, we need to define what it means to do so. We introduce the notion of the *primary component* of a relation for this purpose. The primary component may be thought of as the argument about which the relation tells us something, or as the one around which the relation is defined. Each relation has a single primary component argument specified when the relation is introduced. In our specifications, we will mark the primary component argument of each relation we introduce with $*$. For some relations, such as the typing relation we will introduce as part of our example language, the argument that should be chosen as the primary component is clear. Typing is *about* the expression being typed, rather than the typing context or type it has, and thus the expression is the primary component. In other situations, which argument should be the primary component is not as clear. For example, in a language with a subtyping relation $\tau_1 <: \tau_2$, the relation is truly about the relationship between the two types, not about one over the other. Thus the choice of the primary component is arbitrary.

Because a relation is about its primary component, a rule might change the semantics of an established construct if it can apply to existing constructors of the primary component. This requires the rules an extension module introduces for imported relations to have a term built at the top level by a constructor introduced by the extension module as the primary component of its conclusion. This would ban the above T-Add-Complex rule because the primary component of its conclusion, $add(e_1, e_2)$, is built at the top by the imported *add* constructor. Then extension modules are free to define imported relations for the new constructs they introduce, which have no prior definitions, but not to redefine them for

imported constructs.

This requirement may seem too strict. For example, suppose a module introduces types a and c and a subtyping relation $<:$. An extension to this module wants to introduce a type b such that $a <: b$ and $b <: c$. One of the rules defining these relationships will be disallowed, as the primary component of the subtyping relation must be either the first or second argument, and one of the rules would have an imported construct as that argument. However, the host language designer can recognize this is a problem and instead introduce *two* subtyping relations, $<:_1$ and $<:_2$, the former having the first argument as its primary component and the latter the second as its primary component, along with rules that $\tau_1 <:_1 \tau_2$ holds when $\tau_1 <:_2 \tau_2$ holds and vice versa. Then the extension's desired rules can be encoded in the appropriate relations as $a <:_2 b$ and $b <:_1 c$, and the two relations together form the one subtyping relation desired. This approach of auxiliary relations can be applied to many situations where the restriction on extension-introduced rules would prevent useful extensions from being written otherwise.

Figure 2.2 gives the full set of relations $\mathscr{R}^H$ for the host language along with a selection of the rules defining its relations. The full specification for our example language, including the host language and extensions, can be found in Appendix A. The host language introduces relations for looking up types ($\mathit{lkpTy}(\Gamma^*, n, t)$) and values ($\mathit{lkpVal}(\gamma^*, n, e)$) in their respective types of contexts, as well as a relation indicating a name is *not* present in a typing context ($\mathit{notBoundTy}(\Gamma^*, n)$), all defined as expected. Note the context is the primary component of all these relations, marked with $*$.

Figure 2.2 also contains a selection of the host language's rules for relations defined over expressions and statements. Each of these relations is designed to tell us something *about* the expression or statement over which it is defined, so the statement or expression is the primary component of each relation. The *value* predicate ($\mathit{value}(e^*)$) identifies some expression forms as values. We also have a relation $\mathit{vars}(e^*, ns)$ relating an expression to the set of variables occurring in it. The host language also introduces typing relations for both expressions and statements. Expression typing, written $\Gamma \vdash e^* : ty$, indicates the expression e has the type ty under the typing context Γ assigning types to names. Statement typing,

$$\mathscr{R}^H = \{lkpTy(\Gamma^*, n, ty),\quad notBoundTy(\Gamma^*, n),\quad lkpVal(\gamma^*, n, e),\quad value(e^*),\\ vars(e^*, 2^n),\quad \Gamma \vdash e^* : ty,\quad \Gamma \vdash s^*, \Gamma,\quad \gamma \vdash e^* \Downarrow e,\quad (\gamma,\ s^*) \Downarrow \gamma\}$$

$\mathbb{R}^H$ includes

$\boxed{value(e^*)}$

$$\frac{}{value(intlit(i))}\ \text{V-Int} \qquad \frac{}{value(true)}\ \text{V-True} \qquad \frac{}{value(false)}\ \text{V-False}$$

$\boxed{\Gamma \vdash e^* : ty}$

$$\frac{\Gamma \vdash e_1 : int \qquad \Gamma \vdash e_2 : int}{\Gamma \vdash eq(e_1, e_2) : bool}\ \text{T-Eq} \qquad \frac{\Gamma \vdash e : bool}{\Gamma \vdash not(e) : bool}\ \text{T-Not}$$

$$\frac{\Gamma \vdash e_1 : int \qquad \Gamma \vdash e_2 : int}{\Gamma \vdash add(e_1, e_2) : int}\ \text{T-Add}$$

$\boxed{\Gamma \vdash s^*, \Gamma}$

$$\frac{\Gamma \vdash e : ty \qquad notBoundTy(\Gamma, n)}{\Gamma \vdash decl(n, ty, e), consty(n, ty, \Gamma)}\ \text{TS-Decl} \qquad \frac{\Gamma \vdash s_1, \Gamma' \qquad \Gamma' \vdash s_2, \Gamma''}{\Gamma \vdash seq(s_1, s_2), \Gamma''}\ \text{TS-Seq}$$

$$\frac{\Gamma \vdash e : bool \qquad \Gamma \vdash s_1, \Gamma' \qquad \Gamma \vdash s_1, \Gamma''}{\Gamma \vdash ifte(e, s_1, s_2), \Gamma}\ \text{TS-If}$$

$\boxed{vars(e^*, ns)}$

$$\frac{}{vars(var(n), \{n\})}\ \text{VR-Var} \qquad \frac{}{vars(intlit(i), \varnothing)}\ \text{VR-Int}$$

$$\frac{vars(e_1, vr_1) \qquad vars(e_2, vr_2)}{vars(eq(e_1, e_2), (vr_1 \cup vr_2))}\ \text{VR-Eq}$$

$\boxed{\gamma \vdash e^* \Downarrow e}$

$$\frac{\gamma \vdash e_1 \Downarrow v_1 \qquad \gamma \vdash e_2 \Downarrow v_2 \qquad v_1 = v_2}{\gamma \vdash eq(e_1, e_2) \Downarrow true}\ \text{E-Eq-T}$$

$$\frac{\gamma \vdash e_1 \Downarrow v_1 \qquad \gamma \vdash e_2 \Downarrow v_2 \qquad v_1 \neq v_2}{\gamma \vdash eq(e_1, e_2) \Downarrow false}\ \text{E-Eq-F}$$

$$\frac{\gamma \vdash e_1 \Downarrow intlit(i_1) \qquad \gamma \vdash e_2 \Downarrow intlit(i_2) \qquad plus(i_1, i_2, i)}{\gamma \vdash add(e_1, e_2) \Downarrow intlit(i)}\ \text{E-Add} \qquad \frac{lkpVal(\gamma, n, v)}{\gamma \vdash var(n) \Downarrow v}\ \text{E-Var}$$

$\boxed{(\gamma,\ s^*) \Downarrow \gamma}$

$$\frac{(\gamma,\ s_1) \Downarrow \gamma' \qquad (\gamma',\ s_2) \Downarrow \gamma''}{(\gamma,\ seq(s_1, s_2)) \Downarrow \gamma''}\ \text{X-Seq} \qquad \frac{\gamma \vdash e \Downarrow true \qquad (\gamma,\ s_1) \Downarrow \gamma'}{(\gamma,\ ifte(e, s_1, s_2)) \Downarrow \gamma'}\ \text{X-If-T}$$

Figure 2.2: Relations introduced by the host language ($\mathscr{R}^H$) and selected rules defining them ($\mathbb{R}^H$)

written $\Gamma \vdash s^*, \Gamma'$, indicates the statement s is well-typed with the initial typing context Γ, with any new variable bindings it adds in Γ'. The statement typing relation may be different in a couple ways than expected due to the simplicity of our language. In the TS-Decl rule, we check the name being declared is not already present in the typing context ($notBoundTy(\Gamma, n)$). The TS-If rule types both branches, but then throws away the new variables declared in each, like the branches are scopes in C or Java.

Finally, the host language introduces relations for big-step evaluation for expressions and statements. Expression evaluation, written $\gamma \vdash e^* \Downarrow e'$, specifies how the primary component expression evaluates to a value e', using values from γ for any variables evaluated in it. We see this in E-Var, where a variable evaluates to the value associated with its name in γ. Statement evaluation, written $(\gamma,\ s^*) \Downarrow \gamma'$, specifies how the statement updates the evaluation context, starting with an initial evaluation context γ and updating it through variable declarations and assignments to get γ'.

The relations introduced by the host language are also defined by the extensions for their new syntax, with rules found in Figures 2.3 and 2.4 for the list extension and security extension, respectively. Note in these figures that our extensions obey the requirement that they may only introduce rules for imported relations where the primary component of the conclusion is built by a new constructor. Note also this requirement prevents the extensions from giving new rules for relations with non-extensible primary components, such as $lkpTy(\Gamma, n, ty)$, because new constructors of the primary components cannot be given by extensions. The rules introduced by our extensions are as expected. The list extension identifies the *nil* and $cons(e_1, e_2)$ expression forms as values, with the latter only a value if both its sub-expressions are values, and all other rules it introduces are as expected. Similarly, the security extension's *secdecl* construct is treated in the same way as the host language's *decl* for typing and evaluation.

The security extension also introduces and defines its own relations intended to check programs are secure, that is, that information from private variables cannot leak to public ones in evaluation. In addition to a relation ($lkpSec(\Sigma^*, n, sl)$) for looking up the security level of a variable in a security context, it introduces three other relations. Some rules for

$\mathbb{R}^L$ includes

$\boxed{value(e^*)}$

$$\frac{}{value(nil)}\ \text{V-Nil} \qquad \frac{value(e_1) \qquad value(e_2)}{value(cons(e_1, e_2))}\ \text{V-Cons}$$

$\boxed{\Gamma \vdash e^* : ty}$

$$\frac{\Gamma \vdash e : list(ty)}{\Gamma \vdash null(e) : bool}\ \text{T-Null} \qquad \frac{\Gamma \vdash e : list(ty)}{\Gamma \vdash tail(e) : list(ty)}\ \text{T-Tail}$$

$\boxed{vars(e^*, 2^n)}$

$$\frac{vars(e_1, vr_1) \qquad vars(e_2, vr_2)}{vars(cons(e_1, e_2), (vr_1 \cup vr_2))}\ \text{VR-Cons} \qquad \frac{vars(e, vr)}{vars(head(e), vr)}\ \text{VR-Head}$$

$\boxed{\gamma \vdash e^* \Downarrow e}$

$$\frac{\gamma \vdash e_1 \Downarrow v_1 \qquad \gamma \vdash e_2 \Downarrow v_2}{\gamma \vdash cons(e_1, e_2) \Downarrow cons(v_1, v_2)}\ \text{E-Cons} \qquad \frac{\gamma \vdash e \Downarrow cons(v_1, v_2)}{\gamma \vdash head(e) \Downarrow v_1}\ \text{E-Head}$$

$\boxed{\Gamma \vdash s^*, \Gamma}$

$$\frac{\Gamma \vdash e : list(ty) \qquad lkpTy(\Gamma, n_{hd}, ty) \qquad lkpTy(\Gamma, n_{tl}, list(ty))}{\Gamma \vdash splitlist(n_{hd}, n_{tl}, e), \Gamma}\ \text{TS-Splitlist}$$

$\boxed{(\gamma,\ s^*) \Downarrow \gamma}$

$$\frac{\begin{array}{c}\gamma \vdash e \Downarrow cons(v_1, v_2) \\ n_{hd} \neq n_{tl} \qquad update(\gamma, n_{hd}, v_1, \gamma') \qquad update(\gamma', n_{tl}, v_2, \gamma'')\end{array}}{(\gamma,\ splitlist(n_{hd}, n_{tl}, e)) \Downarrow \gamma''}\ \text{X-Splitlist}$$

Figure 2.3: Selected rules given by the list extension ($\mathbb{R}^L$) for the relations introduced by the host language ($\mathscr{R}^H$)

$\mathbb{R}^S$ includes

$\boxed{\Gamma \vdash s^*, \Gamma}$

$$\frac{\Gamma \vdash e : ty \qquad notBoundTy(\Gamma, n)}{\Gamma \vdash secdecl(n, ty, sl, e), consty(n, ty, \Gamma)}\ \text{TS-SECDECL}$$

$\boxed{(\gamma,\ s^*) \Downarrow \gamma}$

$$\frac{\gamma \vdash e \Downarrow v}{(\gamma,\ secdecl(n, ty, sl, e)) \Downarrow consval(n, v, \gamma)}\ \text{X-SECDECL}$$

Figure 2.4: Rules given by the security extension ($\mathbb{R}^S$) for the relations introduced by the host language ($\mathscr{R}^H$)

these are found in Figure 2.5. The first is $join(sl^*, sl, sl)$, which relates two security levels to the more sensitive of them, that is, the one for information that is more confidential. This relation is truly about the relationship between the security levels, so there is no clear choice for the primary component, and we arbitrarily choose one. Note the J-PRIVATE-R rule has a meta-variable for the primary component of its conclusion. Because this rule is introduced by the same module that introduces the relation, it does not need to give a specific constructor for the primary component.

The other two relations are the *level* relation for expressions and the *secure* relation for statements. The *level* relation, written $\Sigma \vdash level(e^*, sl)$, determines the maximum security level of information used to evaluate the expression, where variables are assigned security levels by the security context Σ; in practice, since our language only contains the *public* and *private* security levels, this determines whether e contains any variables assigned *private* in Σ or not. Some rules for this relation are found in Figure 2.5; as seen there, constants are considered *public* because they have no private information (L-INT), variables have the level assigned them in the security context (L-VAR), and compound expressions have the maximum (determined by *join*) confidentiality level of their sub-expressions (L-EQ).

For statements, $\Sigma\ sl \vdash secure(s^*, \Sigma')$ is intended to hold only if the statement will not leak private information to public variables, with public and private variables determined by Σ. As with statement typing, Σ' contains updated security bindings from the declarations

$$\mathscr{R}^S = \{lkpSec(\Sigma^*, n, sl), \quad join(sl^*, sl, sl), \quad \Sigma \vdash level(e^*, sl), \quad \Sigma\ sl \vdash secure(s^*, \Sigma)\}$$

$\mathbb{R}^S$ includes

$\boxed{join(sl*, sl, sl)}$

$$\frac{}{join(public, public, public)}\ \text{J-PUBLIC} \qquad \frac{}{join(private, \ell, private)}\ \text{J-PRIVATE-L}$$

$$\frac{}{join(\ell, private, private)}\ \text{J-PRIVATE-R}$$

$\boxed{\Sigma \vdash level(e^*, sl)}$

$$\frac{}{\Sigma \vdash level(intlit(i), public)}\ \text{L-INT} \qquad \frac{lkpSec(\Sigma, n, \ell)}{\Sigma \vdash level(var(n), \ell)}\ \text{L-VAR}$$

$$\frac{\Sigma \vdash level(e_1, \ell_1) \qquad \Sigma \vdash level(e_2, \ell_2) \qquad join(\ell_1, \ell_2, \ell)}{\Sigma \vdash level(eq(e_1, e_2), \ell)}\ \text{L-EQ}$$

$\boxed{\Sigma\ sl \vdash secure(s^*, \Sigma)}$

$$\frac{\Sigma\ \ell \vdash secure(s_1, \Sigma') \qquad \Sigma'\ \ell \vdash secure(s_2, \Sigma'')}{\Sigma\ \ell \vdash secure(seq(s_1, s_2), \Sigma'')}\ \text{S-SEQ}$$

$$\frac{\Sigma \vdash level(e, public)}{\Sigma\ public \vdash secure(decl(n, ty, e), conssec(n, public, \Sigma))}\ \text{S-DECL}$$

$$\frac{\Sigma \vdash level(e, \ell) \qquad lkpSec(\Sigma, n, private)}{\Sigma\ \ell' \vdash secure(assign(n, e), \Sigma)}\ \text{S-ASSIGN-PRIVATE}$$

$$\frac{\Sigma \vdash level(e, public) \qquad lkpSec(\Sigma, n, public)}{\Sigma\ public \vdash secure(assign(n, e), \Sigma)}\ \text{S-ASSIGN-PUBLIC}$$

$$\frac{\Sigma \vdash level(e, \ell) \qquad join(\ell', \ell, \ell'') \qquad \Sigma\ \ell'' \vdash secure(s, \Sigma')}{\Sigma\ \ell' \vdash secure(while(e, s), \Sigma)}\ \text{S-WHILE}$$

$$\frac{\Sigma \vdash level(e, \ell)}{\Sigma\ \ell' \vdash secure(secdecl(n, ty, private, e), conssec(n, private, \Sigma))}\ \text{S-SECDECL-PRIVATE}$$

$$\frac{\Sigma \vdash level(e, public)}{\Sigma\ public \vdash secure(secdecl(n, ty, public, e), conssec(n, public, \Sigma))}\ \text{S-SECDECL-PUBLIC}$$

Figure 2.5: Selected rules in $\mathbb{R}^S$ for security relations introduced by S ($\mathscr{R}^S$).

within a statement. The *sl* argument gives the security level of the context in which the statement is executed. Taking a whole-program view, the context level for a statement is determined by whether a private variable may have been used in branching to reach the statement. For example, in the S-While rule, the context level for the body is the more sensitive of the current context level and the condition's level. If the condition uses a private variable, then taking a public action like assigning to a public variable in the loop body can tell a user whether the loop was taken or not, and thus something about the value of the private variable in the condition. Consider the S-Assign-Private and S-Assign-Public rules in Figure 2.5. We may assign to a private variable regardless of the context, as assigning to a private variable cannot leak information because observers cannot see its value. However, to assign to a public variable, the expression the value of which is being assigned to the variable must not contain private variables, and the context in which the assignment is made must be public as well.

2.1.3 Viewing Extensions via Projection

In our example language, we have both an extension introducing analyses on expressions and statements to check confidential information does not flow into public variables from whence it may be leaked to the person running a program, with rules defining these analyses, and an extension introducing new expression and statement forms for lists. If these two extensions are used together, how are the security analyses defined for the new list constructs? The security extension could not introduce rules for them, as it did not know the constructs. The list extension could not introduce rules for them, as it did not know the relations. The composition cannot be useful if the security analyses cannot be derived for the list constructs, as no program using lists could be determined to be secure. We would then need to choose between checking a program's information flow and using lists.

To help define new relations for other modules' new constructs, our framework includes a method for viewing extension constructs through similar non-extension constructs using *projection relations*, introduced in $\mathscr{T}$ in the module-defining tuple $\langle \mathbb{B}, \mathscr{C}, \mathbb{C}, \mathscr{R}, \mathbb{R}, \mathscr{T}, \mathbb{T}, \mathbb{S} \rangle$.

Projection relations in $\mathscr{T}^H$: $\{e : proj_e(e, e),\ s : proj_s(s, s),\ ty : proj_{ty}(ty, ty)\}$

Projection relation rules in $\mathbb{T}^L$:

$$\frac{}{proj_e(null(e), e)}\ \text{P-Null} \qquad \frac{}{proj_e(head(e), e)}\ \text{P-Head}$$

$$\frac{}{proj_e(tail(e), e)}\ \text{P-Tail} \qquad \frac{}{proj_e(nil, true)}\ \text{P-Nil}$$

$$\frac{}{proj_e(cons(e_1, e_2), eq(e_1, e_2))}\ \text{P-Cons} \qquad \frac{}{proj_{ty}(list(ty), ty)}\ \text{P-List}$$

$$\frac{n_{hd} \neq n_{tl}}{\begin{array}{l} proj_s(splitlist(n_{hd}, n_{tl}, e), \\ \quad seq(seq(assign(n_{hd}, e), assign(n_{tl}, tail(var(n_{hd})))), \\ \quad\quad assign(n_{hd}, head(var(n_{hd}))))) \end{array}}\ \text{P-Splitlist}$$

Projection relation rule in $\mathbb{T}^S$:

$$\frac{}{proj_s(secdecl(n, ty, sl, e), decl(n, ty, e))}\ \text{P-Secdecl}$$

Figure 2.6: Projection relations and rules defining them for the host language H and extensions L and S

Projection relations relate terms built at the top level by constructors introduced by extension modules to their *projections* in the same category. A term's projection can be thought of as an approximation of the original term and its semantics. Each relation in $\mathscr{T}^M$ is mapped by $\mathscr{T}^M$ to a syntax category in $\mathscr{C}^M$. Rules defining projection relations for new constructors are found in the $\mathbb{T}$ element of the module tuple.

Earlier we noted that some syntax categories are extensible and some are not. The existence of a projection relation for a syntax category is what makes that category extensible, permitting extension modules to introduce new constructors for it. If $\mathscr{T}^M$ does not include a projection relation for a syntax category it introduces, modules building on M may not contribute new constructors building expressions in that category.

Figure 2.6 shows the projection relations introduced by the host language and rules defining them for the constructs from the list and security extensions. We only have projection relations for expressions, statements, and types as the other categories introduced by the host language (n, i, Γ, and γ) are not extensible. The projection relations in our

example modules take the most basic form for projection relations, being binary relations relating a term and its projection. This is due to the simplicity of our language. In general, projection relations are $(n+2)$-ary relations for some $n \geq 0$. In addition to the term and its projection, they can have arguments that support projection by giving information about the context in which the term being projected occurs. For example, a projection relation might rely on a typing context so expressions can use type information to determine to what to project, or a list of names known at the current point in a program so statements can generate fresh names to use as temporary variables. Both of these uses will be seen in example languages in Chapter 7.

Some of the projections given by the list extension may seem strange; for example, $\mathit{null}(e)$ projecting to e seems to lose some important information. Our framework for language extensibility does not require anything particular of projections beyond well-typedness, that a term projects to a term of the same syntax category. Whether the chosen term is a reasonable one is left to the judgment of the language designer. The reasoning framework we will present in Chapter 4 lets us define what we expect of projections by introducing properties we expect to be true of them. These properties specify what is important for projections to preserve about the original term. Our list projections obey the properties that will be introduced by the language, so the apparent loss of information is fine; the properties specify this lost information is not important.

When an extension module introduces a relation over an imported syntax category, it also introduces a *default rule*, found in the $\mathbb{S}$ portion of the module, that will apply only to terms in the primary component position built by constructs from other extension modules that could not have the relation defined directly. The only requirement for the form of a default rule is that its conclusion have a meta-variable as its primary component. That being said, these rules often have a form similar to

$$\frac{\mathit{proj}_\tau(x_i, y) \qquad R(x_1, \ldots x_{i-1}, y, x_{i+1}, \ldots x_n)}{R(x_1, \ldots x_{i-1}, x_i, x_{i+1}, \ldots x_n)}$$

$\mathbb{S}^S$ includes

$$\frac{proj_e(e, e') \qquad \Sigma \vdash level(e', \ell)}{\Sigma \vdash level(e, \ell)} \text{ L-DEFAULT}$$

$$\frac{proj_s(s, s') \qquad \Sigma\ \ell \vdash secure(s', \Sigma')}{\Sigma\ \ell \vdash secure(s, \Sigma')} \text{ S-DEFAULT}$$

Figure 2.7: Default rules given by the security extension ($\mathbb{S}^S$) for its relations

$$\frac{proj_e(cons(e_1, e_2), e') \qquad \Sigma \vdash level(e', \ell)}{\Sigma \vdash level(cons(e_1, e_2), \ell)} \text{ L-DEFAULT}(cons)$$

$$\frac{proj_e(null(e), e') \qquad \Sigma \vdash level(e', \ell)}{\Sigma \vdash level(null(e), \ell)} \text{ L-DEFAULT}(null)$$

$$\frac{proj_s(splitlist(n_{hd}, n_{tl}, e), s') \qquad \Sigma\ \ell \vdash secure(s', \Sigma')}{\Sigma\ \ell \vdash secure(splitlist(n_{hd}, n_{tl}, e), \Sigma')} \text{ S-DEFAULT}(splitlist)$$

Figure 2.8: Default rules from the security extension instantiated for some list constructs

where the primary component argument to R, x_i, is projected and R is derived for its projection. With a default rule like this, the definition of R for a term built by a constructor from another extension module, one unrelated to the module defining R, is given by copying the definition from its projection.

We see this form of default rule with the security extension's analyses, found in Figure 2.7. In a composed language with both the security and list extensions, the definitions of the *level* and *secure* relations would be given for the list extension's expression and statement constructors by instantiating the L-DEFAULT and S-DEFAULT rules for them. To instantiate a rule for a constructor, we replace the primary component argument of the conclusion with a term built by the constructor with fresh meta-variables as arguments. Figure 2.8 shows some examples of instantiating the security extension's default rules for constructs from the list extension. The L-DEFAULT(*cons*) rule instantiates the default rule for *level* for the list extension's *cons* constructor, and the L-DEFAULT(*null*) rule instantiates it for the list extension's *null* constructor. The S-DEFAULT(*splitlist*) rule similarly instantiates the S-DEFAULT rule for the list extension's *splitlist* construct.

While the form given above is a common one, and new semantic relations are often

defined well by using their projections, sometimes it is useful to use the projection in a different way, or not to use the projection. As an example, we add an optimization extension to perform constant folding, with its relations and some rules shown in Figure 2.9. It does not introduce any new syntax, only new relations and their definitions. The main relations are $opt_e(e^*, e)$ and $opt_s(s^*, s)$, relating expressions and statements to their optimized forms, with auxiliary relations $notInt(e^*)$ and $notBool(e^*)$ that hold when the expression is not constructed by *intlit* for the former or *true* or *false* for the latter, that is, the expression is not a value form of the specified type. The default rule for optimizing statements has the same form as above, but for expressions, our default rule optimizes an expression to itself (rule OE-Default). Similarly, the default rules for the auxiliary relations define them to hold for any unknown construct as these cannot be built by the specified constructors.

Consider optimizing an expression

$$add(add(intlit(3), intlit(4)), head(cons(add(intlit(3), intlit(4)), nil)))$$

The optimization uses the OE-Add-O-2 rule. The first sub-expression of the top-level symbol, $add(intlit(3), intlit(4))$, optimizes to $intlit(7)$ using the OE-Add-I and OE-Int rules. The second sub-expression, $head(cons(add(intlit(3), intlit(4)), nil))$, uses the OE-Default rule instantiated for the list extension's *head* construct. This optimizes the whole expression to itself. The $notInt(head(cons(add(intlit(3), intlit(4)), nil)))$ premise for the OE-Add-O-2 rule is derived by the NI-Default rule, also instantiated for *head*. Thus the whole expression optimizes to

$$add(intlit(7), head(cons(add(intlit(3), intlit(4)), nil)))$$

This result seems unsatisfying because it is clear to us that the whole expression must always evaluate to $intlit(14)$. However, the optimization extension cannot more usefully optimize the list extension's constructs. It does not know they exist, so it cannot provide rules specific to them, including ones that would optimize only sub-expressions. Thus the best it can do

$$\mathscr{R}^O = \{opt_e(e^*, e), \quad opt_s(s^*, s), \quad notInt(e^*), \quad notBool(e^*)\}$$

$\mathbb{R}^O$ includes

$\boxed{opt_e(e^*, e)}$

$$\frac{}{opt_e(intlit(i), intlit(i))}\ \text{OE-Int} \qquad \frac{}{opt_e(var(n), var(n))}\ \text{OE-Var}$$

$$\frac{opt_e(e_1, intlit(i_1)) \qquad opt_e(e_2, intlit(i_2)) \qquad plus(i_1, i_2, i)}{opt_e(add(e_1, e_2), intlit(i))}\ \text{OE-Add-I}$$

$$\frac{opt_e(e_1, e_1') \qquad opt_e(e_2, e_2') \qquad notInt(e_1')}{opt_e(add(e_1, e_2), add(e_1', e_2'))}\ \text{OE-Add-O-1}$$

$$\frac{opt_e(e_1, e_1') \qquad opt_e(e_2, e_2') \qquad notInt(e_2')}{opt_e(add(e_1, e_2), add(e_1', e_2'))}\ \text{OE-Add-O-2}$$

$\boxed{opt_s(s^*, s)}$

$$\frac{opt_e(e, e')}{opt_s(assign(n, e), assign(n, e'))}\ \text{OS-Assign}$$

$$\frac{opt_e(c, false)}{opt_s(while(c, b), skip)}\ \text{OS-While-F} \qquad \frac{opt_e(c, c') \qquad opt_s(b, b') \qquad c' \neq false}{opt_s(while(c, b), while(c', b'))}\ \text{OS-While-O}$$

$\mathbb{S}^O$ includes

$$\frac{}{opt_e(e, e)}\ \text{OE-Default} \qquad \frac{proj_s(s, s') \qquad opt_s(s', s'')}{opt_s(s, s'')}\ \text{OS-Default}$$

$$\frac{}{notInt(e)}\ \text{NI-Default} \qquad \frac{}{notBool(e)}\ \text{NB-Default}$$

Figure 2.9: Relations introduced by the optimize extension ($\mathscr{R}^O$) and selected rules defining them ($\mathbb{R}^O$ and $\mathbb{S}^O$)

is not eliminate them, as we see in the OE-Default rule. While this may not reduce all constant expressions, it also means it does not rewrite away any constructs introduced by other extensions, which would lose their unique semantics. As we will see in future chapters, in a property introduced by the optimization extension, its measure of correctness is that optimization does not change the values in evaluation, a goal accomplished for constructs from unrelated extensions by using the OE-Default rule.

Our example language consists of the host language and three extensions, the list extension, the security extension, and the optimization extension. While none of our extensions build on others, we note that the ability to do so that is part of our framework is useful, particularly for more complex extensions. An intermediate extension can provide a level of abstraction somewhere between the new module's functionality and that of the host language, making it easier to ensure the projections of terms are related to the original terms in expected ways. For example, it would be natural to build an extension introducing matrices on top of one introducing lists, with matrices projecting to nested lists and matrix operations projecting to sequences of list operations. Allowing modules to build on one another in arbitrary ways also permits combining the features introduced by independent modules, as we see with some extensions to the AbleC extensible version of the C programming language [17]. This has an extension introducing regular expressions and matching strings against them, and another extension, independent of the first, introducing algebraic datatypes (*i.e.*, datatypes like those in a functional language such as OCaml) along with pattern matching on them. A third extension module builds on both of these, adding a construct to include regular expressions in patterns and adding checking if a string matches a regular expression to the evaluation of pattern matching. This is something that neither of the other extensions could do on their own.

2.2 Well-Formedness

We have well-formedness requirements to which language modules must adhere. A language library is well-formed only if each module in it is also well-formed.

One simple well-formedness requirement, which we will not mention further, is each rule a module introduces must be well-typed. The specifications of constructors in $\mathbb{C}$ and relations in $\mathscr{R}$ and $\mathscr{T}$ impose typing constraints by specifying the syntax categories for their arguments, and these must be maintained. As part of this, we assume all syntax categories, constructors, and relations that are part of a module specification are known, being introduced either by the module being specified or one it imports. In typing, we assume each syntax category, constructor, and relation introduced by each module has a name unique across all modules in a language library, and thus can be uniquely identified. In practice, this can be accomplished by qualifying the name of each construct with the name of the module introducing it, as is accomplished in the Java ecosystem.

Well-formedness for modules has several parts, some of which have been mentioned above. To summarize them formally, in one place:

- Each projection relation in $\mathscr{T}^M$ is mapped by $\mathscr{T}^M$ to a syntax category in $\mathscr{C}^M$, and only one projection relation may map to each category (*i.e.*, the mapping is injective). Thus each syntax category has at most one projection relation. Furthermore, each projection relation, projecting a category C, takes at least two arguments of type C, one for the projecting term and one for its projection.

- For each constructor $c \in \mathbb{C}^M$ building expressions in a syntax category $C \in \mathscr{C}^N$, $N \neq M$, C has a corresponding projection relation in $\mathscr{T}^N$, that is, each new constructor of a syntax category from a built-on module builds an extensible category.

- Each rule in $\mathbb{R}^M$ defining a relation from $\mathscr{R}^N$, $N \neq M$, has as the primary component argument of its conclusion a term built at the top level by a constructor from $\mathscr{C}^M$, that is, each rule defining a relation from a built-on module defines it for a new constructor.

- Each rule in $\mathbb{T}^M$ defines a relation from $\mathscr{T}^N$, $N \neq M$, and has as its primary component a term built at the top level by a constructor from $\mathscr{C}^M$, that is, each rule defining a projection relation from a built-on module defines it for a new constructor.

- There is a unique default rule in $\mathbb{S}^M$ for each relation in $\mathscr{R}^M$ where the primary

component category of the relation is a syntactic category from $\mathscr{C}^N$, $N \neq M$, that is, the primary component category is from a built-on module.

- Each rule in $\mathbb{S}^M$ has as the primary component of its conclusion a variable.
- If a module M builds on a module N ($N \in \mathbb{B}^M$) and N builds on a module O ($O \in \mathbb{B}^N$), then M must also build on O ($O \in \mathbb{B}^M$). This requires the builds-on relationships to be transitive.
- A module cannot build on itself ($M \notin \mathbb{B}^M$).

The final two requirements disallow circularity in module dependencies in a language library, as no individual module may introduce a circularity to it (the final requirement) and all its dependencies must be known when it is written (the penultimate requirement).

Note we do not require a rule defining the projection for each new constructor building expressions in an imported syntax category. Choosing not to do so for a new construct limits extensibility, as the relations introduced by other extensions may then not be defined for the new construct when the default rule depends on the projection. This is, however, a choice an extension writer may make, being aware of its possible negative effects.

Well-formedness of an individual module can be checked independently of modules other than the ones on which it builds. As noted earlier, language libraries start with some small set of modules, with developers writing new extension modules building on those already published as part of the library. This allows developers to check their modules are well-formed before contributing them to the library, as all the modules on which a new one builds are already known, and a new module can be checked against them. Thus we may assume all modules in a library are well-formed, as developers can ensure their modules are well-formed before contributing them.

2.3 Module Composition

An extensible language by itself is not useful, as it cannot be used to write a program, unless it can be turned into a full language with defined, non-extensible syntax and semantics. A

full language is a 4-tuple $\langle \mathscr{C}, \mathbb{C}, \mathscr{R}, \mathbb{R} \rangle$ containing the language's syntax categories, syntax constructors, relations, and rules defining those relations. We will define the composition of modules to form a full language as the language defined by one module and the modules on which it builds.

Part of the language of a module will be instantiations of default rules as discussed in Section 2.1.3. As discussed there, default rules are instantiated for constructs from unrelated extensions. The set of instantiated rules is how the definitions of extension-introduced relations are completed in composition. A relation, such as the security extension's *level* relation, is defined for constructs from extensions unknown to it, such as the list extension, using the *default* rule; this rule is the default in that any construct for which no other definition could have been written is given one using it.

Definition 2.1 (Instantiated default rule set)**.** *The instantiated default rule set of a set of modules $\mathscr{S}$, written $\mathbb{R}^{\mathbb{S}}(\mathscr{S})$, instantiates each default rule from one module in $\mathscr{S}$ with each constructor from each unrelated module in $\mathscr{S}$. It does this by replacing the meta-variable in the primary-component position of the default rule's conclusion with a term whose top-level symbol is the new constructor, and that has fresh meta-variables for the arguments. Formally, the set $\mathbb{R}^{\mathbb{S}}(\mathscr{S})$ is*

$$
\begin{aligned}
\{r[c(\overline{y})/x_i] \mid\ & M_j \in \mathscr{S} \ \wedge\ M_k \in \mathscr{S} \ \wedge \\
& M_j \notin \mathbb{B}^{M_k} \ \wedge\ M_k \notin \mathbb{B}^{M_j} \ \wedge\ M_j \neq M_k \\
& r \in \mathbb{S}^{M_j} \ \wedge\ c \in \mathbb{C}^{M_k} \ \wedge \\
& \mathit{defines}(r) = R \ \wedge\ \mathit{pc}(r) = x_i \ \wedge\ \mathit{fresh}(\overline{y}, r) \ \wedge \\
& \mathit{pc}(R) = C \ \wedge\ \mathit{category}(c) = C\}
\end{aligned}
$$

where	$\mathit{defines}(r)$	*identifies the relation defined by rule r*
	$\mathit{pc}(r)$	*identifies the primary component term of the conclusion of rule r*
	$\mathit{fresh}(\overline{y}, r)$	*means $\overline{y}$ is a set of variables where none appear in rule r*
	$\mathit{pc}(R)$	*identifies the primary component category of relation R*

$category(c)$ *identifies the category for which expressions are built by constructor* c

$r[c(\overline{y})/x_i]$ *means replacing each occurrence of* x_i *in* r *with* $c(\overline{y})$

Note the instantiation is only carried out for well-typed pairs of rules and constructors, where the primary component category of the relation being defined by the default rule matches the one for which the constructor builds expressions.

The language of a module includes all syntax categories, syntax constructors, and relations, both projection and non-projection relations, of the module and the modules on which it builds. It also includes all the rules defining the projection and non-projection relations, as well as the instantiated default rule set for the set of known modules.

Definition 2.2 (Language of a module)**.** *Let* M *be a module in a language library. Let* $\mathscr{S}$ *be its builds-on set and itself,* $\mathbb{B}^M \cup \{M\}$*. Let*

- $\mathscr{C} = \bigcup_{N \in \mathscr{S}} \mathscr{C}^N$
- $\mathbb{C} = \bigcup_{N \in \mathscr{S}} \mathbb{C}^N$
- $\mathscr{R} = \bigcup_{N \in \mathscr{S}} (\mathscr{R}^N \cup \mathscr{T}^N)$
- $\mathscr{R} = \mathbb{R}^{\mathbb{S}}(\mathscr{S}) \cup \bigcup_{N \in \mathscr{S}} (\mathbb{R}^N \cup \mathbb{T}^N)$

The language of M*, written* $Lang(M)$*, is the full language* $\langle \mathscr{C}, \mathbb{C}, \mathscr{R}, \mathbb{R} \rangle$*.*

We do not define the composition of modules into full languages beyond the language of a single module. This is not limiting, in comparison to a definition of composition explicitly for a set of modules, as the composition of multiple modules can be accomplished by creating a module simply building on them without introducing anything new. To get the composition of our example language's modules H, L, O, and S, we can build a dummy module D where $\mathbb{B}^D = \{H, L, O, S\}$ and all the other sets in the module tuple are empty. Then $Lang(D)$ is the composition of all other modules in our example language. Thus defining only the language of a single module covers any general composition case. The builds-on structure of all the modules in our example language, including D, is shown in

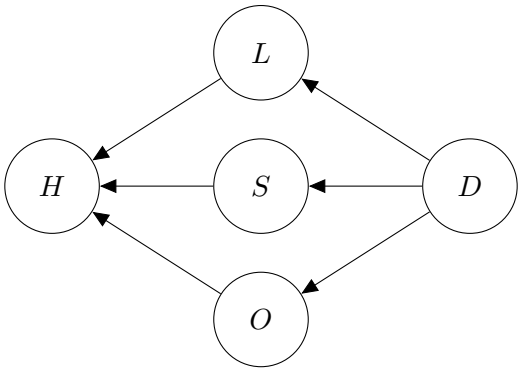


Figure 2.10: Structure of modules in our example language

Figure 2.10, with arrows pointing from a module building on another to the one on which it builds. We elide the arrow from D to H, as that builds-on relationship can be inferred from the other relationships due to the requirement that builds-on relationships must be transitive for well-formed modules. For example, because S both builds on H and is built-on by D, we know D must also build on H.

A summary of the D module and its language is given in Figure 2.11. The language of D has all the syntax categories introduced by any of the modules, so it includes expressions e, statements s, and security levels sl, among others. It also has all syntax constructors introduced by each of the modules, as well as all relations, including projection relations, introduced by all the modules. Finally, it includes the rules from each module, including rules defining projections, such as the T-Add and P-Cons rules, and instantiated default rules ($\mathbb{R}^{\mathbb{S}}(\{H, L, O, S\})$) replacing the primary component meta-variable of the default rule with terms built by new constructors from other extensions. Three instantiated default rules are shown in Figure 2.11 as examples; note the full language contains many more rules arising from instantiation. The default rule L-Default is shown instantiated for the *cons* and *null* constructors (L-Default(*cons*) and L-Default(*null*)), and the OE-Default rule is shown instantiated for the *cons* constructor (OE-Default(*cons*)). The instantiation of other default rules and other constructors is similar.

$D = \langle \{H, L, O, S\}, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing \rangle$

$Lang(D) = \langle \mathscr{C}, \mathbb{C}, \mathscr{R}, \mathbb{R} \rangle$

$\mathscr{C} = \{s, e, ty, n, i, \Gamma, \gamma, sl, \Sigma\}$

$\mathbb{C} = \{skip, decl, assign, seq, ifte, while, var, intlit, true, false, add, eq, gt, not,$
$int, bool, nilty, consty, nilval, consval, nil, cons, null, head, tail, list, splitlist,$
$public, private, nilsec, conssec, secdecl\}$

$\mathscr{R} = \{lkpTy(\Gamma, n, ty), \quad notBoundTy(\Gamma, n), \quad lkpVal(\gamma, n, e), \quad value(e), \quad vars(e, 2^n),$
$\Gamma \vdash e : ty, \quad \Gamma \vdash s, \Gamma, \quad \gamma \vdash e \Downarrow e, \quad (\gamma,\ s) \Downarrow \gamma \quad lkpSec(\Sigma, n, sl), \quad join(sl, sl, sl),$
$\Sigma \vdash level(e, sl), \quad \Sigma\ sl \vdash secure(s, \Sigma), \quad opt_e(e, e), \quad opt_s(s, s), \quad notInt(e),$
$notBool(e), \quad proj_e(e, e), \quad proj_s(s, s), \quad proj_{ty}(ty, ty), \quad proj_{sl}(sl, sl)\}$

$\mathbb{R}$ includes

$$\frac{\Gamma \vdash e_1 : int \qquad \Gamma \vdash e_2 : int}{\Gamma \vdash add(e_1, e_2) : int}\ \text{T-Add}$$

$$\frac{}{proj_e(cons(e_1, e_2), eq(e_1, e_2))}\ \text{P-Cons}$$

$$\frac{proj_e(cons(e_1, e_2), e') \qquad \Sigma \vdash level(e', \ell)}{\Sigma \vdash level(cons(e_1, e_2), \ell)}\ \text{L-Default}(cons)$$

$$\frac{proj_e(null(e), e') \qquad \Sigma \vdash level(e', \ell)}{\Sigma \vdash level(null(e), \ell)}\ \text{L-Default}(null)$$

$$\frac{}{opt_e(cons(e_1, e_2), cons(e_1, e_2))}\ \text{OE-Default}(cons)$$

Figure 2.11: Summary of the module D and its language $Lang(D)$ showing its syntax, relations, and selected rules

2.4 Comparison with Silver's Extensibility Framework

As mentioned earlier, the language extensibility framework we present here is based on the model for extensibility used by the Silver attribute grammar system [18, 38]. Our framework, as presented in this chapter, is nearly identical, but reformulated for rule-based specifications. The logic we will use for our reasoning, discussed in the next chapter, supports rule-based specifications of systems quite directly, whereas we would need to develop a more complex encoding to use attribute grammars.

To discuss how we have molded Silver's framework to our own setting, we must first understand the basics of attribute grammars. Attribute grammars define language semantics by associating semantic attributes with nodes in a syntax tree, syntax trees being built by productions of an underlying grammar. The values of attributes are determined by equations associated with productions. For example, a production for an addition expression might have an equation for an attribute giving the expression's type that checks both of its sub-expressions have an integer type, and then specify its own type as an integer type as well.

As in our framework, Silver allows modules to introduce syntax and semantics, the latter in the form of attributes and equations defining them, with extensions similarly extending existing syntax and semantics. It also limits how extensions may extend the definitions of existing semantics, as we do. Our restriction is that new rules defining imported relations must each have a new constructor, introduced by the extension introducing the rule, as the primary component of the conclusion. Silver has a corresponding requirement to prevent extensions from modifying the semantics of existing constructs. Extensions may only introduce new equations for imported attributes associated with new productions (*i.e.*, new syntax constructors). As is made clear by this correspondence, our notion of a primary component is inspired by the concept of the nonterminal on which an attribute occurs in attribute grammars.

Silver does not have a concept of default rules as we use. Instead it uses a concept called *forwarding* [39]. When a production does not have an associated equation defining

an attribute, it copies the attribute's value from its *forward*, Silver's term for what we introduce as projections. Forwarding is similar to our default rules, as it is how extension-introduced attributes are defined for constructs from unrelated extensions. However, it being able only to *copy* the value from the forward, or projection, can be limiting. As we saw with the OE-Default rule from the optimization extension, which optimizes expressions built by constructs from unknown extensions to themselves, copying may not always give the definitions we want. Sometimes we want this kind of freedom to define our relations for unknown constructs in ways that are specific to the relation being defined rather than Silver's one-size-fits-all approach.

By building our extensibility model on Silver's extensibility model, we know the language framework for which we develop our reasoning framework is a useful one in practice. A number of extensible languages have been developed in Silver. The most significant one is the aforementioned AbleC [17], an extensible version of the C programming language. AbleC has a number of interesting extensions, such as one introducing algebraic datatypes and pattern matching, and another introducing Prolog-style constructs for logic programming. The remainder of this thesis is dedicated to showing how we can make this framework for extensible languages more useful by allowing modules to introduce interesting metatheory along with interesting syntax and semantics.

Chapter 3

A Logic for Reasoning about Language Properties

In proving our reasoning framework is sound, that proofs written by individual modules ensure metatheoretic properties will hold for any composed language, we will need to analyze the structure of proofs. To this end, we choose to write our metatheoretic proofs in the logic called $\mathcal{G}$ [12], which is the basis for the Abella proof assistant [2]. $\mathcal{G}$ provides support for reasoning about definitions over terms, making it easy to reason about the rule-based definitions of relations in languages written using the extensibility framework from the previous chapter. Our arguments for the soundness of our reasoning framework will take the form of constructing metatheoretic proofs in $\mathcal{G}$ in the context of a composed language using parts of the metatheoretic proofs written by modules.

We first introduce the logic $\mathcal{G}$ in Section 3.1. We then give an example of proving a formula using $\mathcal{G}$ in Section 3.2. After that, we describe the encoding of languages into $\mathcal{G}$ definitions in Section 3.3. Finally, in Section 3.4 we describe the form of properties and their proofs that we will consider in the next chapter.

3.1 The Logic $\mathcal{G}$

The logic $\mathcal{G}$ is based on the simply-typed lambda calculus and includes support for notions of binding; however, we are only interested here in a first-order version of it, and elide the non-first-order portions of it, including this support for bindings. Proofs are written in a context including a set of sorts, constructors of those sorts, and predicate symbols. Terms are constructed from these constructors, and atomic formulas from terms by using predicate symbols. Formulas are constructed from atomic formulas; logical constants $\top$ and

$\bot$; logical connectives $\wedge$, $\vee$, and $\supset$; and universal and existential quantification. Formally, quantification is written as $\forall x : \tau.F$ for universal quantification and as $\exists x : \tau.F$ where τ is a sort, with repeated quantification being written as $\forall x_1 : \tau_1 \ldots \forall x_n : \tau_n.F$ for universal quantification, and similarly for existential quantification. For brevity's sake, we generally assume the sort of a bound variable can be determined from its use in the formula and condense repeated quantifications into a shorthand using a single quantifier to represent the sequence. Then the above formula can be written as $\forall x_1, \ldots, x_n.F$ with the understanding it stands for the longer form with appropriate sorts for each bound variable. Formulas differing only in the names of variables, that is, formulas where renaming their variables renders them identical, are treated as identical in $\mathcal{G}$.

The proof rules of $\mathcal{G}$ will rely on notions of variable substitutions and unification, so we define these before presenting the proof rules.

Definition 3.1 (Substitution)**.** *A substitution is a finite sequence of variables $x_1, \ldots, x_n$ and terms $t_1, \ldots, t_n$ that are pairwise of the same type. Such a substitution is written as $\{\langle x_1, t_1\rangle, \ldots, \langle x_n, t_n\rangle\}$. We say its domain is the set of variables $\{x_1, \ldots, x_n\}$ and its range is the set of terms $\{t_1, \ldots, t_n\}$. Applying a substitution θ to a term t or a formula F, written $t[\theta]$ and $F[\theta]$, replaces each occurrence of a variable in the domain of θ with the corresponding term. In formulas, quantified variables must be renamed to avoid capturing variables. The composition of two substitutions, written $\theta_2 \circ \theta_1$, is a substitution such that $e[\theta_2 \circ \theta_1] = e[\theta_1][\theta_2]$ for any e.*

Definition 3.2 (Unification)**.** *A unification problem is a set of pairs of terms or atomic formulas, written $\{\langle e_1^1, e_1^2\rangle, \ldots, \langle e_n^1, e_n^2\rangle\}$. A unifier for a unification problem is a substitution θ such that, for each pair $\langle e_i^1, e_i^2\rangle$ in the unification problem, $e_i^1[\theta] = e_i^2[\theta]$. A problem is solvable if a unifier exists. A most general unifier, or mgu, for a unification problem is a unifier θ where any other unifier θ' can be written as θ composed with another substitution ρ ($\theta' = \rho \circ \theta$). It is a known fact that, in our setting, a unification problem that is solvable has a most general unifier.*

The logic $\mathcal{G}$ is presented in the style of a sequent calculus. Sequents have the form

$$\frac{}{\Sigma : \Gamma, A \longrightarrow A}\ id,\ A \text{ atomic} \qquad \frac{\Sigma : \Gamma \longrightarrow B \qquad \Sigma : B, \Delta \longrightarrow C}{\Sigma : \Gamma, \Delta \longrightarrow C}\ cut \qquad \frac{\Sigma : \Gamma, B, B \longrightarrow C}{\Sigma : \Gamma, B \longrightarrow C}\ c\mathcal{L}$$

$$\frac{}{\Sigma : \Gamma, \bot \longrightarrow C}\ \bot\mathcal{L} \qquad\qquad \frac{}{\Sigma : \Gamma \longrightarrow \top}\ \top\mathcal{R}$$

$$\frac{\Sigma : \Gamma, B \longrightarrow C \qquad \Sigma : \Gamma, D \longrightarrow C}{\Sigma : \Gamma, B \vee D \longrightarrow C}\ \vee\mathcal{L} \qquad\qquad \frac{\Sigma : \Gamma \longrightarrow B_i}{\Sigma : \Gamma \longrightarrow B_1 \vee B_2}\ \vee\mathcal{R}_i,\ i \in \{1, 2\}$$

$$\frac{\Sigma : \Gamma, B_i \longrightarrow C}{\Sigma : \Gamma, B_1 \wedge B_2 \longrightarrow C}\ \wedge\mathcal{L}_i,\ i \in \{1, 2\} \qquad\qquad \frac{\Sigma : \Gamma \longrightarrow B \qquad \Sigma : \Gamma \longrightarrow C}{\Sigma : \Gamma \longrightarrow B \wedge C}\ \wedge\mathcal{R}$$

$$\frac{\Sigma : \Gamma \longrightarrow B \qquad \Sigma : \Gamma, D \longrightarrow C}{\Sigma : \Gamma, B \supset D \longrightarrow C}\ \supset\mathcal{L} \qquad\qquad \frac{\Sigma : \Gamma, B \longrightarrow C}{\Sigma : \Gamma \longrightarrow B \supset C}\ \supset\mathcal{R}$$

$$\frac{\Sigma : \Gamma, B[\{\langle x, t\rangle\}] \longrightarrow C}{\Sigma : \Gamma, \forall x : \tau.B \longrightarrow C}\ \forall\mathcal{L} \qquad\qquad \frac{(\Sigma, x : \tau) : \Gamma \longrightarrow B}{\Sigma : \Gamma \longrightarrow \forall x : \tau.B}\ \forall\mathcal{R},\ x \notin \Sigma$$

$$\frac{(\Sigma, x : \tau) : \Gamma, B \longrightarrow C}{\Sigma : \Gamma, \exists x : \tau.B \longrightarrow C}\ \exists\mathcal{L},\ x \notin \Sigma \qquad\qquad \frac{\Sigma : \Gamma \longrightarrow B[\{\langle x, t\rangle\}]}{\Sigma : \Gamma \longrightarrow \exists x : \tau.B}\ \exists\mathcal{R}$$

In the $\forall\mathcal{L}$ and $\exists\mathcal{R}$ rules, the term t must be well-typed with respect to the sequent's eigenvariable context Σ and the global constant context

Figure 3.1: Rules in $\mathcal{G}$ for the logical symbols

$\Sigma : \Gamma \longrightarrow F$ where Σ is a collection of typed variables called the *eigenvariable context* representing universal quantification over the sequent, Γ is a multiset of *assumption formulas* or *hypotheses*, and F is the sequent's *conclusion* or *goal formula*. As we drop the types on quantified variables for brevity's sake when the type is determinable from the formula, so we drop the types on variables in the eigenvariable context when their types can be determined from the rest of the sequent, writing it as simply a list of variables. A sequent is well-formed if F and all the formulas in Γ are well-typed with respect to the eigenvariable context for the sequent and the types of the constants in the context. We will henceforth assume all sequents are well-formed.

Proofs in $\mathcal{G}$ are derivations of sequents using proof rules interpreting the meaning of those sequents. The core rules are shown in Figure 3.1. Most of these rules are clear, so

we comment only on those we believe may need some explanation. First, the $\forall\mathcal{L}$ and $\exists\mathcal{R}$ rules instantiate the quantified variable with a term that must be of the same sort as the variable. Furthermore, the instantiating term may be constructed only from the variables in the eigenvariable context Σ and the set of constructors in the context in which the proof is being written. The other set of rules on which we comment are the $\forall\mathcal{R}$ and $\exists\mathcal{L}$ rules. These both require introducing new variables. These new variables must be fresh to the eigenvariable context; if they are not fresh, the variables in the formula may be renamed to make them fresh.

The logic $\mathcal{G}$ allows atomic predicates to be given an interpretation based on fixed-point definitions. In addition to being parameterized by a context of sorts, constructors, and predicate symbols, $\mathcal{G}$ is also parameterized by a set $\mathcal{D}$ of *definitional clauses* defining the predicate symbols. Each clause has the form $\forall\overline{x}.H \triangleq B$ where $\overline{x}$ is a sequence of variables called the clause's *binder*; H is an atomic formula, called the clause's *head*; and B is an arbitrary formula,[1] called the clause's *body*. As with formulas, clauses differing only in variable names are equivalent, and we call them *variants* of each other. We say a clause is *named away* from a collection of variables Σ if its binder contains names distinct from those in Σ.

Each clause gives a portion of the definition of an atomic predicate, with the full definition being given by all the clauses where the head is built by the same predicate. This interpretation of the clauses is given by the proof rules in Figure 3.2 for using definitions in rules. The application of a substitution to an eigenvariable context, as seen in these rules, means removing from it the variables in the domain of the substitution and adding those in the terms in its range. The *def*$\mathcal{R}$ rule allows us to prove an atomic formula by showing one of the clauses defining the predicate building it applies. To do so, the atomic goal formula must be an instance of the clause's head, and we must prove the clause's body, appropriately instantiated. The *def*$\mathcal{L}$ rule goes the other direction, allowing us to prove the sequent by proving the conclusion holds regardless of how the atomic assumption formula was derived.

[1]Definitional clauses are expected to obey certain stratification conditions to make the logic consistent. We will assume all clauses obey this, noting the simple rule form permitted in our language specification framework is not complex enough to write rules breaking the stratification conditions.

$$\frac{\Sigma[\theta] : \Gamma \longrightarrow B[\theta]}{\Sigma : \Gamma \longrightarrow A}\ def\mathcal{R}$$

$\forall \overline{x}.H \triangleq B$ is a variant of a clause in $\mathcal{D}$ and θ is a substitution such that $A = H[\theta]$

$$\frac{\{\ \Sigma[\theta] : \Gamma[\theta], B[\theta] \longrightarrow C[\theta] \mid \forall \overline{x}.H \triangleq B \text{ is a variant of a clause in } \mathcal{D} \text{ named away from } \Sigma \text{ and } \theta \text{ is an mgu for } \{\langle A, H\rangle\}\ \}}{\Sigma : \Gamma, A \longrightarrow C}\ def\mathcal{L}$$

Figure 3.2: Rules for introducing atomic formulas based on a definition $\mathcal{D}$

Because $\mathcal{G}$ uses a fixed-point definition of predicates, we know it was derived using one of the clauses defining it, so we need to consider only a most-general form of it using each clause. This rule encodes a case-analysis style of reasoning that shall feature prominently in our modular reasoning framework. Each premise sequent of this rule is based on one definition clause defining the relation that is the top-level predicate for A. The definition clause is named away from the eigenvariables already appearing in the sequent to avoid inadvertent capture of names, and the head of the clause is unified with the atomic formula. If the unification succeeds, the atomic formula could have been derived using the rule, and the proof rule requires showing the conclusion follows if that were the case.

$\mathcal{G}$ allows us to use definitions inductively. To do so, we associate a measure with atomic formulas, marking whether a formula is the original size with which we started or smaller than it, with the induction hypothesis being valid only for the latter.[2] Measures are written as annotations on formulas, with the annotation for the original size being $@^i$ and the annotation for a smaller derivation being $*^i$, where i is a strictly positive natural number, standing for i repetitions of the annotation (*e.g.*, by $*^3$, we mean an annotation $***$). Annotations are introduced to a sequent by the use of the ind^i_m rule, seen in Figure 3.3. This allows us to assume the goal formula for smaller versions of one of its premises, annotated with $*^i$, while proving the formula with the same premise, annotated with $@^i$ to mark it as the original size, in order to prove the original sequent.

[2]For simplicity in presenting our reasoning framework, we use the implementation of induction in Abella [2], a proof assistant for writing proofs in $\mathcal{G}$, rather than the one in the original treatment of the logic [37].

$$\frac{}{\Sigma : \Gamma, A^{*^i} \longrightarrow A^{*^i}}\ id^{**} \qquad \frac{}{\Sigma : \Gamma, A^{@^i} \longrightarrow A^{@^i}}\ id^{@@}$$

$$\frac{}{\Sigma : \Gamma, A^{*^i} \longrightarrow A^{@^i}}\ id^{*@} \qquad \frac{}{\Sigma : \Gamma, A^{*^i} \longrightarrow A}\ id^{*} \qquad \frac{}{\Sigma : \Gamma, A^{@^i} \longrightarrow A}\ id^{@}$$

$$\frac{\{\ \Sigma[\theta] : \Gamma[\theta], (B[\theta])^{*^i} \longrightarrow C[\theta] \mid \forall \overline{x}.H \triangleq B \text{ is a variant of a clause in } \mathcal{D} \text{ named away from } \Sigma \text{ and } \theta \text{ is an mgu for } \{\langle A, H\rangle\}\ \}}{\Sigma : \Gamma, A^{*^i} \longrightarrow C}\ def\mathcal{L}^{*^i}$$

$$\frac{\{\ \Sigma[\theta] : \Gamma[\theta], (B[\theta])^{*^i} \longrightarrow C[\theta] \mid \forall \overline{x}.H \triangleq B \text{ is a variant of a clause in } \mathcal{D} \text{ named away from } \Sigma \text{ and } \theta \text{ is an mgu for } \{\langle A, H\rangle\}\ \}}{\Sigma : \Gamma, A^{@^i} \longrightarrow C}\ def\mathcal{L}^{@^i}$$

$$\frac{\Sigma : \Gamma, \forall \overline{x_1}.F_1 \supset \cdots \supset \forall \overline{x_m}.A^{*^i} \supset F \longrightarrow \forall \overline{x_1}.F_1 \supset \cdots \supset \forall \overline{x_m}.A^{@^i} \supset F}{\Sigma : \Gamma \longrightarrow \forall \overline{x_1}.F_1 \supset \cdots \supset \forall \overline{x_m}.A \supset F}\ ind^i_m,\ A \text{ is atomic}$$

Annotations of the form $*^i$ and $@^i$ must not already appear in the conclusion sequent

Figure 3.3: Induction rule and associated rules for annotated formulas

We have special versions of the *id* and *def*$\mathcal{L}$ rules to support annotations, also shown in Figure 3.3; the regular *id*, *def*$\mathcal{L}$, and *def*$\mathcal{R}$ rules do not apply to annotated atomic formulas. The special versions of the *id* rule ensure annotated formulas match sufficiently with the goal formulas they are used to prove. In interpreting the meanings of these rules, the measures can be interpreted as "no larger than"; thus A^{*^i} can be used in the place of $A^{@^i}$ because it is not larger than the measure represented by $@^i$, and it can be used to prove A without an annotation because un-annotated formulas do not have size restrictions. The special versions of the *def*$\mathcal{L}$ rule show that unfolding reduces the measure assigned to a formula, either from $@^i$ to $*^i$ or maintaining $*^i$. We write $B[\theta]^{*^i}$ where $B[\theta]$ may not be an atomic formula to represent appropriately passing the annotation down to the atomic formulas within it.

We finish this section by defining what we mean by applying a substitution to a proof and showing that substitution in a sequent does not invalidate previous proofs.

Definition 3.3 (Proof substitution)**.** *Let π be a proof in $\mathcal{G}$ and let θ be a substitution. Applying θ to π, written $\pi[\theta]$, is defined recursively on the structure of π by these two rules:*

1. *If the last rule in π is the def$\mathcal{L}$ rule or one of its variants, suppose the conclusion of this rule was $\Sigma : \Gamma, A \longrightarrow F$ where A is a possibly-annotated atomic formula and it has premise proofs $\pi_1, \ldots, \pi_n$. Each π_i arises from a clause of the form $\forall \overline{x_i}.H_i \triangleq B_i$ where there is an mgu θ_i for the unification problem $\{\langle A, H_i\rangle\}$. If $\{\langle A[\theta], H_i\rangle\}$ has a unifier, there must be a substitution ρ_i such that $\theta_i \circ \rho_i$ is an mgu for it. The substituted proof uses the same variant of the def$\mathcal{L}$ rule with the conclusion sequent being $\Sigma[\theta] : \Gamma[\theta], A[\theta] \longrightarrow F[\theta]$ and premise proofs $\pi_i[\rho_i]$ for each i where $\{\langle A[\theta], H_i\rangle\}$ is solvable.*

2. *If the last rule in π is not the def$\mathcal{L}$ rule or one of its variants, suppose the sequent in its conclusion is $\Sigma : \Gamma \longrightarrow F$ and it has premise proofs $\pi_1, \ldots, \pi_n$. Then the substituted proof $\pi[\theta]$ uses the same rule, but with its conclusion being $\Sigma[\theta] : \Gamma[\theta] \longrightarrow F[\theta]$ and with premise proofs $\pi_1[\theta], \ldots, \pi_n[\theta]$.*

Theorem 3.4 (Proof substitution validity)**.** *Let $\mathcal{S}$ be a sequent $\Sigma : \Gamma \longrightarrow F$. If $\mathcal{S}$ has a proof π of height h, then, for any substitution θ, $\pi[\theta]$ is a proof of $\Sigma[\theta] : \Gamma[\theta] \longrightarrow F[\theta]$ of height at most h.*

Proof. We proceed by induction on the structure of π. It is clear the transformation for rules other than *def*$\mathcal{R}$ or *def*$\mathcal{L}$ and its variants has the desired property. The transformation for *def*$\mathcal{R}$ also produces a proof that is no taller than the original because, for a clause $\forall \overline{x}.H \triangleq B$, if F is an atomic formula such that $F = H[\rho]$, then we know $F[\theta] = H[\rho][\theta]$, and the two premise proofs we need are π and $\pi[\theta]$, respectively.

For the *def*$\mathcal{L}$ rule and its variants, as explained in Definition 3.3, some subset of the same clauses unify, and the induction hypothesis shows the substituted versions of their original proofs prove the new sequents. It is also clear this is the full set of clauses that unify; clauses that could not unify with A cannot unify with $A[\theta]$. We note the removal of some premise proofs might shorten the proof tree, as it might prune the tallest branches, and so the resulting proof is no taller than the original. ■

3.2 An Example of Reasoning

To illustrate writing proofs in $\mathcal{G}$, consider proving that appending two lists is unique. We have lists with two constructors, the *nil* constructor representing an empty list and the cons constructor, written infix as ::, creating a list from a head element and a tail list. The *append* relation has two clauses, defining appending two lists in the standard way:

$$\forall l.append(nil, l, l) \triangleq \top$$
$$\forall h, t, l, t'.append(h :: t, l, h :: t') \triangleq append(t, l, t')$$

We will also use a defined equality relation, written infix as $=$, that relates two terms that are exactly the same:

$$\forall a.a = a \triangleq \top$$

This gives a standard definition of equality.

We can state that the *append* relation is unique in its first two arguments as a formula:

$$\forall l_1, l_2, r_1, r_2.append(l_1, l_2, r_1) \supset append(l_1, l_2, r_2) \supset r_1 = r_2$$

To prove this, we prove the sequent

$$\varnothing : \varnothing \longrightarrow \forall l_1, l_2, r_1, r_2.append(l_1, l_2, r_1) \supset append(l_1, l_2, r_2) \supset r_1 = r_2$$

This specifies we are proving this formula with no pre-existing eigenvariables and no pre-existing hypotheses. We will induct on the first derivation of *append* ($append(l_1, l_2, r_1)$), so we use the ind_1^1 rule. This gives us the sequent

$$\varnothing : \forall l_1, l_2, r_1, r_2.append(l_1, l_2, r_1)^* \supset append(l_1, l_2, r_2) \supset r_1 = r_2 \longrightarrow$$
$$\forall l_1, l_2, r_1, r_2.append(l_1, l_2, r_1)^{@} \supset append(l_1, l_2, r_2) \supset r_1 = r_2$$

The first premise in the conclusion is annotated with @ to mark it as having the original

size of the derivation, and the induction hypothesis has its first premise annotated with $*$ to mark it as being applicable only to a smaller derivation than the original one. We can use the $\forall\mathcal{R}$ rule four times and the $\supset\mathcal{R}$ rule twice, so we need to prove

$$l_1, l_2, r_1, r_2 : (\forall l_1, l_2, r_1, r_2.\mathit{append}(l_1, l_2, r_1)^* \supset \mathit{append}(l_1, l_2, r_2) \supset r_1 = r_2),$$
$$\mathit{append}(l_1, l_2, r_1)^{@}, \mathit{append}(l_1, l_2, r_2) \longrightarrow r_1 = r_2$$

To do so, we can use the $\mathit{def}\mathcal{L}^{@}$ rule to analyze the $\mathit{append}(l_1, l_2, r_1)^{@}$ hypothesis. Both definition clauses unify with this, giving us two premise sequents to prove.

The first premise sequent is for the first definition clause, where l_1 is nil and the other two lists must be the same, so r_1 is unified with l_2:

$$l_2, r_2 : (\forall l_1, l_2, r_1, r_2.\mathit{append}(l_1, l_2, r_1)^* \supset \mathit{append}(l_1, l_2, r_2) \supset r_1 = r_2),$$
$$\top, \mathit{append}(\mathit{nil}, l_2, r_2) \longrightarrow l_2 = r_2$$

We also have an added assumption of $\top$ from the body of the clause. We can apply the un-annotated $\mathit{def}\mathcal{L}$ rule to $\mathit{append}(\mathit{nil}, l_2, r_2)$. Only the first definition clause can apply to this assumption because its first argument list is nil, so we have one premise sequent:

$$l_2 : (\forall l_1, l_2, r_1, r_2.\mathit{append}(l_1, l_2, r_1)^* \supset \mathit{append}(l_1, l_2, r_2) \supset r_1 = r_2), \top, \top \longrightarrow l_2 = l_2$$

Because this definition clause requires the latter two arguments to be the same, we have unified r_2 and l_2, and we can complete the proof of this sequent with the $\mathit{def}\mathcal{R}$ rule and the clause defining equality, and the $\top\mathcal{R}$ rule to prove its body.

The other premise sequent from the initial case analysis is for the other definition clause, where the first list's top-level symbol is the cons constructor:

$$h, t, l_2, t', r_2 : (\forall l_1, l_2, r_1, r_2.\mathit{append}(l_1, l_2, r_1)^* \supset \mathit{append}(l_1, l_2, r_2) \supset r_1 = r_2),$$
$$\mathit{append}(t, l_2, t')^*, \mathit{append}(h :: t, l_2, r_2) \longrightarrow h :: t' = r_2$$

The original l_1 list is unified with $h :: t$ in this sequent, and r_1 is $h :: t'$ for some list t'. We also have a hypothesis $append(t, l_2, t')^*$ from the body of the clause, annotated with $*$ because, being a sub-derivation of the original one, it is a derivation smaller in size than the original. We want to use the induction hypothesis with this premise, but to do so we need to break down the second derivation of *append* as we did in the previous case, which we can do using the un-annotated *def*$\mathcal{L}$ rule. As in the prior case, only one clause applies, so we have one premise sequent, where r_2 is unified with $h :: t''$ for some list t'':

$$h, t, l_2, t', t'' : (\forall l_1, l_2, r_1, r_2. append(l_1, l_2, r_1)^* \supset append(l_1, l_2, r_2) \supset r_1 = r_2), \\ append(t, l_2, t')^*, append(t, l_2, t'') \longrightarrow h :: t' = h :: t''$$

Having two derivations of *append* with the same first two arguments, one of which is annotated with $*$, we can use the induction hypothesis. To do so, we apply the $\forall\mathcal{L}$ rule four times and the $\supset\mathcal{L}$ rule twice, once each with the annotated id^{**} rule and un-annotated *id* rule, the former for the annotated premise and the latter for the un-annotated premise. This gives us a sequent

$$h, t, l_2, t', t'' : (\forall l_1, l_2, r_1, r_2. append(l_1, l_2, r_1)^* \supset append(l_1, l_2, r_2) \supset r_1 = r_2), \\ append(t, l_2, t')^*, append(t, l_2, t''), t' = t'' \longrightarrow h :: t' = h :: t''$$

The hypothesis from this application, $t' = t''$, can be analyzed with the un-annotated *def*$\mathcal{L}$ rule, showing that t' and t'' are the same and unifying them, so the conclusion of the premise sequent is $h :: t' = h :: t'$. Then the *def*$\mathcal{R}$ and $\top\mathcal{R}$ rules can be used to complete the proof of the sequent as in the previous case. Because we have proven all the sequents needed, this also completes the proof of the original formula overall.

3.3 Encoding Languages into the Logic

Recall from Section 2.3 that a full language is a 4-tuple $\langle \mathscr{C}, \mathbb{C}, \mathscr{R}, \mathbb{R} \rangle$ containing the syntax categories, syntax constructors, relations, and rules for the full language. It is this 4-tuple we encode to $\mathcal{G}$ creating the context of sorts, constructors, and predicates, as well as the set $\mathcal{D}$ of clauses defining the predicates. We will say we are reasoning in the context of a language $Lang(M)$ when we are using the context and set of clauses $\mathcal{D}$ corresponding to the tuple $\langle \mathscr{C}^{Lang(M)}, \mathbb{C}^{Lang(M)}, \mathscr{R}^{Lang(M)}, \mathbb{R}^{Lang(M)} \rangle$.

The first three elements of the tuple become, respectively, the sorts, syntax constructors, and predicate symbols in $\mathcal{G}$. The correspondence is immediate, so we will use the same syntax for terms and relations in the context of $\mathcal{G}$ as we used in Chapter 2.

Each rule in $\mathbb{R}$ becomes a clause in $\mathcal{D}$. For a rule

$$\frac{B_1 \qquad \ldots \qquad B_m}{R(t_1, \ldots, t_n)}$$

in $\mathbb{R}$, there is a corresponding clause

$$\forall \overline{x}.R(t_1, \ldots, t_n) \triangleq \exists \overline{y}.B_1 \wedge \ldots \wedge B_m$$

in $\mathcal{D}$ where $\overline{x}$ contains all the variables in $t_1, \ldots, t_n$ and $\overline{y}$ contains all the variables in $B_1, \ldots, B_m$ not in $\overline{x}$. Furthermore, these are the only rules in $\mathcal{D}$, so there is a perfect correspondence between the clauses used for reasoning and the rules in the language definition.

Consider our example language from the previous chapter. The host language introduces a rule for evaluating addition:

$$\frac{\gamma \vdash e_1 \Downarrow \mathit{intlit}(i_1) \qquad \gamma \vdash e_2 \Downarrow \mathit{intlit}(i_2) \qquad \mathit{plus}(i_1, i_2, i)}{\gamma \vdash \mathit{add}(e_1, e_2) \Downarrow \mathit{intlit}(i)} \text{ E-ADD}$$

We can encode this into a definition clause in $\mathcal{D}$ as

$$\forall \gamma, e_1, e_2, i.(\gamma \vdash \mathit{add}(e_1, e_2) \Downarrow \mathit{intlit}(i)) \triangleq$$
$$\exists i_1, i_2.(\gamma \vdash e_1 \Downarrow \mathit{intlit}(i_1)) \wedge (\gamma \vdash e_2 \Downarrow \mathit{intlit}(i_2)) \wedge \mathit{plus}(i_1, i_2, i)$$

The variables γ, e_1, e_2, and i found in the rule's conclusion are universally quantified at the beginning, over the whole rule, while the variables i_1 and i_2 that occur only in the body are existentially quantified in the body.

We can similarly encode the typing rule for addition, T-Add:

$$\frac{\Gamma \vdash e_1 : \mathit{int} \qquad \Gamma \vdash e_2 : \mathit{int}}{\Gamma \vdash \mathit{add}(e_1, e_2) : \mathit{int}} \text{ T-Add}$$

When translated into a definition clause, this becomes

$$\forall \Gamma, e_1, e_2.(\Gamma \vdash \mathit{add}(e_1, e_2) : \mathit{int}) \triangleq (\Gamma \vdash e_1 : \mathit{int}) \wedge (\Gamma \vdash e_2 : \mathit{int})$$

Unlike in E-Add, we don't have any variables occurring in the premises that aren't in the conclusion, so we have no quantifier in the rule's body.

3.4 Metatheoretic Properties in the Logic

The properties we will prove about encoded languages in our framework will be written as formulas in $\mathcal{G}$ using the vocabulary of the language of some module M, $\mathit{Lang}(M)$. The properties we will consider will have the form

$$\forall \overline{x}.R(\overline{t}) \supset F$$

where R is a relation, $\overline{t}$ is a set of terms given to R as arguments, and F is an arbitrary formula. The terms in $\overline{t}$ are built using the variables in $\overline{x}$ and the constants in $\mathit{Lang}(M)$.

This form is equivalent to the more general form

$$\forall \overline{x_1}.F_1 \supset \forall \overline{x_2}.F_2 \supset \ldots \supset \forall \overline{x_n}.F_n$$

where some F_i, $i \neq n$, is $R(\bar{t})$. We restrict the form for simplicity in discussion to highlight the first premise. In a property of this form, we will call R the *key relation*, and we will orient our proof around its definition.

Consider some examples of properties that are introduced by the modules in our language from Chapter 2. First, the host language introduces the property of type preservation, that if a well-typed expression evaluates to a value with related typing and evaluation contexts, that value has the same type as the expression had:

$$\begin{aligned}&\forall \Gamma, \gamma, e, ty, v.\gamma \vdash e \Downarrow v \supset \Gamma \vdash e : ty \supset \\ &\quad (\forall x, ty_x, v_x.\mathit{lkpTy}(\Gamma, x, ty_x) \supset \mathit{lkpVal}(\gamma, x, v_x) \supset \mathit{nilty} \vdash v_x : ty_x) \supset \mathit{nilty} \vdash v : ty \end{aligned} \quad (3.1)$$

The key relation of this property is expression evaluation.

The optimized evaluation property introduced by the optimization extension module states that optimizing an expression does not change whether it evaluates or the value to which it evaluates:

$$\forall e, e', \gamma, v.\mathit{opt}_e(e, e') \supset \gamma \vdash e \Downarrow v \supset \gamma \vdash e' \Downarrow v \quad (3.2)$$

The key relation of this property is the opt_e relation introduced by the optimization extension module.

Finally, the security extension module introduces a property that its analysis guarantees information from private variables does not leak into public variables in evaluating a

statement:

$$\forall s, \Sigma, sl, \Sigma', \gamma_1, \gamma_1', \gamma_2, \gamma_2'.(\gamma_1,\ s) \Downarrow \gamma_1' \supset (\gamma_2,\ s) \Downarrow \gamma_2' \supset \Sigma\ sl \vdash \mathit{secure}(s, \Sigma') \supset \mathit{eqpublicvals}(\Sigma, \gamma_1, \gamma_2) \supset \mathit{eqpublicvals}(\Sigma', \gamma_1', \gamma_2') \quad (3.3)$$

Here $\mathit{eqpublicvals}(\Sigma, \gamma_1, \gamma_2)$ stands for the formula

$$\forall x.\mathit{lkpSec}(\Sigma, x, \mathit{public}) \supset (\forall v.\mathit{lkpVal}(\gamma_1, x, v) \supset \mathit{lkpVal}(\gamma_2, x, v)) \wedge (\forall v.\mathit{lkpVal}(\gamma_2, x, v) \supset \mathit{lkpVal}(\gamma_1, x, v))$$

This property states that when a statement judged secure is evaluated under two different evaluation contexts that have the same values for all variables considered public but that may differ in values for private variables, the resulting evaluation contexts also have the same values for all public variables. This formalizes the notion of not leaking private information, as it ensures the values of private variables cannot have any effect on the values of public variables. The key relation here is the statement evaluation relation. Note there are two derivations of this relation, evaluating s under γ_1 and γ_2. When discussing the key relation of a property, we mean also the specific derivation of it that is the first premise of the property.

We introduce the *canonical form* of a proof of a metatheoretic property. As we saw in Section 3.2, a proof of a formula F is a proof of a sequent $\varnothing : \varnothing \longrightarrow F$. In the same way, our proofs of properties of the form $\forall \overline{x}.R(\bar{t}) \supset F$ will be proofs of sequents $\varnothing : \varnothing \longrightarrow \forall \overline{x}.R(\bar{t}) \supset F$. The canonical form of a proof ends with some uses of the ind_m^i rule, including a use of the ind_1^i rule to induct on the derivation of the key relation. The proof of the premise sequent of these induction rules applies the $\forall\mathcal{R}$ and $\supset\mathcal{R}$ proof rules to move between eigenvariables and bindings and between premises of the property, including the key relation, and hypotheses of the sequent to prove. These rules have a premise sequent where the key relation's derivation is a hypothesis annotated with $@^i$, allowing us to use the $\mathit{def}\mathcal{L}^{@^i}$ proof rule to carry out the case analysis on the key relation. We call this the

top-level case analysis because it is the only one in the shared part of the proof of the whole property; any other case analyses occur within the cases resulting from this one. The premises of the top-level case analysis are sequents for the various language rules that unify with the derivation of the key relation.

Consider a canonical-form proof of Property 3.2. The sequent we want to prove for this property is

$$\varnothing : \varnothing \longrightarrow \forall e, e', \gamma, v. opt_e(e, e') \supset \gamma \vdash e \Downarrow v \supset \gamma \vdash e' \Downarrow v$$

For the canonical-form proof of this property, we have only one induction, which is on the key relation. We use the $\forall\mathcal{R}$ rule four times to introduce eigenvariables for the four bound variables, and the $\supset\mathcal{R}$ rule twice to introduce the first two premises as hypotheses. The premise sequent of these rules, abbreviating the induction hypothesis as *IH*, is

$$e, e', \gamma, v : IH, opt_e(e, e')^{@}, \gamma \vdash e \Downarrow v \longrightarrow \gamma \vdash e' \Downarrow v$$

To prove this, we use the $def\mathcal{L}^{@}$ rule for the top-level case analysis to analyze the derivation of $opt_e(e, e')$. This leads to a number of premise sequents and proofs of them, one for each language rule defining opt_e.

As the final piece of our discussion of generally proving metatheoretic properties in $\mathcal{G}$, it is common when proving metatheoretic properties of languages to use some properties as lemmas in proving others. The logic $\mathcal{G}$ does not explicitly consider proving properties using a set of known lemmas. However, using previously-proven properties as lemmas is permitted by the *cut* rule. A lemma, stated as a formula F, is assumed to have been proven already, so we have a proof of the sequent $\varnothing : \varnothing \longrightarrow F$. Then we can use the *cut* rule as

$$\dfrac{\dfrac{\vdots}{\Sigma : \varnothing \longrightarrow F} \qquad \dfrac{\vdots}{\Sigma : \Gamma, F \longrightarrow C}}{\Sigma : \Gamma \longrightarrow C}\ cut$$

We reuse the earlier proof of F, renaming it away from Σ if necessary, and then may use the

lemma's formula to prove the conclusion C. Thus it is valid to refer to writing proofs with sets of lemmas that may possibly be used in the proof, as long as those lemmas already have proofs. We will do so throughout the next chapter where we discuss our reasoning framework.

Chapter 4

A Modular Proof Structure for Metatheoretic Properties

Our goal is to permit the metatheory of extensible languages to be developed modularly, with any module in a language library able to introduce new metatheoretic properties and guarantee they hold for any composed language that includes the module. To show a property holds for any composed language, we need a proof of the property for each composed language. Our notion of the canonical form of a proof calls for a top-level case analysis on a premise of the property, the derivation of the key relation. However, knowing only part of the language, the module introducing a property does not know all the cases resulting from this top-level case analysis that may be part of a composed language. Then the author of the module introducing the property cannot write a proof of it for an arbitrary composed language.

We address these issues by distributing proofs across modules and restricting case analysis. First, we distribute the proofs of the cases from the top-level case analysis across modules that know the property, both the one introducing the property and those that build on it. This mitigates part of the problem of the introducing module's limited knowledge of the language for the top-level cases, as other modules that know the property can prove it for the cases they introduce. However, there can be modules that don't know the property but can still contribute rules defining its key relation to a composed language, rules that may create top-level cases, and there can be instantiations of the default rule in a composition that do the same. For example, both the optimization extension's Property 3.2 and the security extension's Property 3.3 will have cases in a composed language for list constructs, even though the list extension does not know either of these properties, nor

do the modules introducing the properties know the list extension. Our novel approach to solving this problem is to have the module introducing a property handle these cases *generically*. This seems difficult to do for interesting properties, as the constructs about which we want to reason generically are by definition not known. However, we know some relations will be defined for them by instantiating default rules, and we know other properties will hold for them. In particular, we introduce the concept of *projection constraints*, a set of properties relating the semantics of extension-introduced constructs with the semantics of their projections. These give a way of understanding unknown constructs introduced by other modules. In a composition, a generic proof can be specialized to the actual rules from unknown modules, since the unknown modules also must have proven the properties on which the generic proof relies.

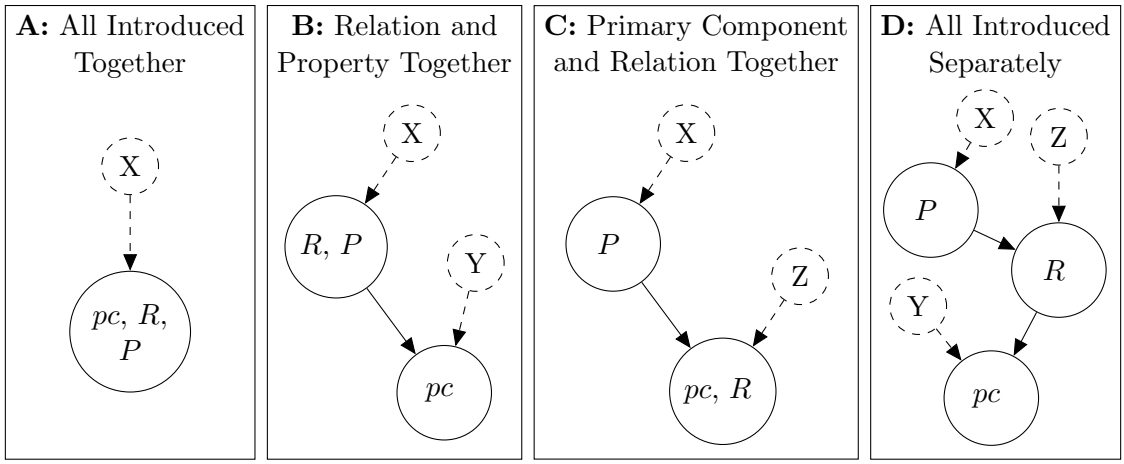
While this approach ensures every case in a composition has a proof for a corresponding case from some module, the limited knowledge available to individual modules is also a problem for using case analysis in proving top-level cases that are known, as other modules might add applicable rules. Specifically, further case analyses within the top-level cases rely on having a closed-world assumption that we do not have. We solve this by limiting the application of case analysis, other than the top-level case analysis, to situations where a *local* closed-world assumption holds. In these situations, all the rules that might be applicable in any composed language are known to the module writing the proof. Then the fact that only a portion of the language is known does not matter, as the known portion contains all the rules that matter. This approach will allow us to compose proofs written by individual modules to form full proofs of properties for composed languages.

We start this chapter by further addressing the difficulties in the modular setting, as well as further laying out our approach to solving the problem, in Section 4.1. As mentioned above, we distribute the proof of a property across the modules that are aware of the property. We discuss the details of writing the proofs for specific modules for both modules importing a property and the module introducing a property in Sections 4.2 and 4.3 respectively. In Section 4.4 we explain how these proofs can be used to create a full proof of the property for any composed language, and prove this full proof will be valid in the context

of the composed language. Specifically, we will prove that, if all modules in a composed language that know a property write valid proofs for it as required of the module based on its relationship to the one introducing the property, whether the module introduces the property or imports it, then there is also a canonical-form proof of the property for the composed language. This proof of the soundness of composition will be constructive, giving us a concrete method for building a full proof of a metatheoretic property for any composed language out of the modular proofs written for each module. Finally, in Section 4.5 we discuss how to extend these ideas to proofs of properties using mutual induction. The ideas in this chapter are a generalization of the approach we presented in a prior paper [27].

4.1 A Structure for Modular Reasoning

To understand our approach to overcoming the difficulties of reasoning in an extensible setting, let us consider how the modules relevant to the introduction of a property can be structured. Specifically, these are the modules introducing the property itself, the property's key relation, and the primary component category of the key relation. Note that the module introducing a property must know its key relation, and the module introducing a relation must know its primary component category, limiting the possible relationships between the modules introducing the elements to four. These are shown in Figure 4.1, along with the ways other modules may build on the modules introducing the three elements. In this figure, arrows point from a module to one on which it builds, with further builds-on relations determined transitively (*e.g.*, the X module in the B scenario also builds on the *pc* module). The modules introducing a property P, its key relation R, and R's primary component *pc* are drawn with solid lines and are labeled with the parts they introduce. Note these modules may build on others introducing other parts of the language than these three key elements; we do not show any such modules to simplify the diagrams. Dashed circles are modules not known to the module where the property is introduced, but which may be written separately from it. These are labeled with categories X, Y, and Z according to which of P, R, and *pc* are known by each module. Specifically, modules labeled X know



Property P, its key relation R, R's primary component category pc

Figure 4.1: Diagrams of the four possible module relations for properties

P, R, and pc; modules labeled Y know pc but not R or P; and modules labeled Z know pc and R but not P.

Consider our three example properties from the previous chapter. Type preservation, Property 3.1, is introduced by the host language module from Chapter 2. Its key relation, expression evaluation, and the primary component category of this relation, e, are also introduced by the host language. Since all three elements are introduced together in the same module, this fits scenario A. The optimization extension's property for the correctness of its opt_e relation for expressions, Property 3.2, has the property and its key relation introduced in the same module. The primary component category e is introduced by the host language module on which it builds, making it fit scenario B. Finally, the security extension introduces Property 3.3 specifying that statements passing its analysis do not leak private information. This has the host language's statement evaluation as its key relation, making it fit scenario C where the property is introduced in a separate module from the one introducing its key relation and the primary component category. We do not have a property for scenario D as our example language does not have an extension module building on another extension, so it does not have the right structure to introduce one.

Our goal is to ensure a proof, specifically a canonical-form proof, will exist for each property introduced by any module for any composed language in which the module is

included. We have each module introducing a property also declare the specific set-up steps for a canonical-form proof for the property (*e.g.*, specifying on which premises to induct in addition to the key relation). This set-up portion of the canonical proof structure has a top-level case analysis on the key relation, creating proof cases for each language rule defining the key relation. To complete the proof for a composed language, we need a proof for the proof case for each rule.

We can break the set of rules defining a property's key relation in a composed language down into four classes, with a rule's class being determined by how the module introducing it relates to the one introducing the property, its key relation, and the key relation's primary component. We list the four classes here, with discussions of how they are handled in proofs and examples of such rules relative to our language's properties to follow:

- **Known rules** are from modules known when the property is introduced, including both the module introducing the property and those on which it builds. These are introduced by modules drawn with solid lines in Figure 4.1.

- **New rules** are from modules that build on the one introducing the property. These are introduced by modules labeled X in Figure 4.1.

- **Instantiated default rules** arise from modules knowing the primary component category but not the key relation. These are associated with modules labeled Y in Figure 4.1.

- **Independent rules** are from modules knowing the key relation but not the property. These are introduced by modules labeled Z in Figure 4.1.

Other modules, ones that do not know even the primary component category, cannot contribute rules to the definition of the key relation in a composed language, so these are all the possibilities.

Our approach distributes the proof cases for language rules across modules. Each module knowing a property will write its own canonical-form proof for it using the portion of the language that it knows. Note that only two types of modules can write such proofs, the one

introducing the property and those that build on it, as these are the only types of modules knowing the property. Keep in mind that whether a module is an introducing module or an extension one is relative to a particular *property*, not a characteristic of the module itself; the same module may take on both roles but for different properties. While these are the only two types of modules knowing a property, only known rules and new rules are known to these modules. Instantiated default rules do not exist until a composition, and independent rules are introduced by modules that are independent of the module introducing the property, and, in general, also independent of those that build on it. While the particulars of these rules are not known, the fact such rules might be part of a composed language is known to the introducing module, and thus we can have it reason *generically* about them as part of the proof it writes. Then the module introducing a property is responsible for proving the cases for known rules, instantiated default rules, and independent rules, while modules extending it are only responsible for proving the cases for new rules they introduce. In the setting of a composed language, the canonical-form proof of a property takes the proof for each case arising from its top-level case analysis from the corresponding case in the proof written by one of the modules included in the composition. We turn now to discussing the particulars of how each class of rules is handled by the proofs written by the modules.

The first class of rules, known rules, are those known to the module introducing a property. These rules are introduced by the module introducing the property or those on which it builds, the modules drawn with solid lines in Figure 4.1. Proof cases for known rules are part of the proof written by the module introducing the property. For the host language's type preservation property, Property 3.1, rules introduced by the host language, such as E-Add for evaluating additions, are of this class because they are known to the host language when it introduces the property. Section 4.3 describes the proofs introducing modules must write, including the cases for known rules.

The next class of rules are those from modules building on the module introducing the property; that is, they are introduced by extensions to the one introducing the property. We call these new rules because the property already exists in another module, but the rules are new relative to it. These are from modules labeled X in Figure 4.1. Proof cases

for new rules are proven by the modules introducing them. For type preservation, rules introduced by our extension modules, such as E-HEAD from the list extension, are of this class because the module introducing the rules imports the property. We discuss proving properties for new rules in Section 4.2.

Our third class of rules are instantiated default rules. Modules labeled Y in Figure 4.1 know only the primary component category, not the key relation, so they cannot introduce rules for the key relation directly. In a composed language, the key relation is defined for new constructs introduced by these modules by instantiating its default rule. For example, the optimization expression's Property 3.2 that fits scenario B in the figure, where the optimization expression is the module introducing R and P, has the list extension as a Y module. Its key relation, opt_e, is defined for the list extension's constructs, such as *cons*, in a composed language by instantiating its default rule for them. The module introducing a property is expected to provide a generic proof that can apply to *any* instantiated default rule. This module knows such cases might exist, depending on the module structure for the property, but not exactly what they are because the constructs are from independent modules. However, it does know the form of the default rule, and that any construct for which it will be instantiated will be introduced by a module not knowing the key relation. This knowledge gives it information it can use to write the requisite generic proof, the details of which are discussed in Section 4.3.

The final class of rules are introduced by modules knowing the property's key relation but not the property itself, those created by modules labeled Z in Figure 4.1. We call these independent rules because they are introduced by modules that are completely independent of the one introducing the property. The security extension's Property 3.3, which corresponds to scenario C in the figure with the security extension being the module introducing P, has the list extension being a Z module relative to it, so the X-SPLITLIST rule from the list extension is a rule of this class relative to the security property. As with instantiated default rules, the module introducing a property knows based on the module structure that Z modules, and thus independent rules, can exist, and is required to provide a generic proof for them as well. Unlike instantiated default rules, it does not know the form these rules

will have. Proving cases for independent rules relies on them being restricted in some ways, a topic we discuss further in Section 4.3.

Thus far we have laid out, at a high level, our scheme for ensuring all proof cases in the composition resulting from a top-level case analysis on the key relation will be handled in the proofs written by the module introducing a property and extensions to it. However, this is not the only problem with case analysis and the closed-world assumption in the extensible setting. Within the proof case for a particular rule resulting from the top-level case analysis, analyzing a premise might prevent the modular proof from being valid in the composed setting, as there could be more rules unifying with it in the composition. The new sub-cases would not have proofs, as they were not known when the modular proof was written. Thus our reasoning framework also circumscribes case analysis within modular proofs, ensuring it is used only when it will not result in missing cases in any composed language. It does this by identifying the situations in which other modules cannot add rules that might be used to derive the atomic formula being analyzed due to our restrictions on the rules extensions may introduce and how languages are composed.

The remainder of this chapter further develops these ideas, describing how proofs are written and circumscribed in both extensions to the module introducing a property and in the introducing module itself. We also prove the soundness of our approach, showing how these proofs can be used to create a proof for any composed language, and how we can extend it to account for mutually-inductive properties.

4.2 Proofs in Extension Modules

We start with proofs written by modules extending the one that introduces a property (*i.e.*, modules labeled X in Figure 4.1). While it might seem strange to start with extensions rather than the module introducing a property, the requirements for extensions are simpler than for those written by modules introducing properties. The next section extends these requirements to the situation for modules introducing properties.

A module has a responsibility to write its own proofs for the properties it imports.

While we define this as a full proof, in practice, this responsibility extends only to the cases for its own, new rules; as noted in the previous section, other modules will be responsible for proving the cases for the rules it imports.

Definition 4.1 (Modular proofs in extension modules)**.** *Let N be a module building on a module M introducing a property $P = \forall \overline{x}.R(\overline{t}) \supset F$ constructed using the vocabulary of $Lang(M)$. A modular proof for P relative to module N is a canonical-form proof for P in the language $Lang(N)$ using as lemmas the properties in $\mathscr{L}$ where the use of the def$\mathcal{L}$ rule and its annotated variants are used to analyze premises of the form $R'(\overline{t'})$ in only the following situations:*

- *the primary component argument of $R'(\overline{t'})$ is built by a constructor or*
- *the primary component category of R' is a non-extensible category.*

These are the only restrictions on proofs.

The limitations placed on case analysis in this definition ensure any analysis, other than the top-level one producing the distributed cases, cannot have any applicable rules in a composed language that are added by other modules unknown at the time the proof is written. Having new rules applicable to a case analysis would lead to holes in the composed proof. We will take up the proof of the sufficiency of these restrictions in Section 4.4, but the intuition behind them is that a module can add a new rule for a relation it imports only for a new constructor of the relation's primary component. Then if a premise's primary component is built by a known constructor or if new constructors, and thus new rules, cannot be added by other modules, all applicable rules are known.

As an example of writing a proof for a case while obeying these restrictions, consider showing Property 3.1, type preservation, for the list module's *head* constructor. If *IH* refers to the induction hypothesis formula

$$\forall \Gamma, \gamma, e, ty, v.(\gamma \vdash e \Downarrow v)^* \supset \Gamma \vdash e : ty \supset$$
$$(\forall x, ty_x, v_x.\mathit{lkpTy}(\Gamma, x, ty_x) \supset \mathit{lkpVal}(\gamma, x, v_x) \supset \mathit{nilty} \vdash v_x : ty_x) \supset \mathit{nilty} \vdash v : ty$$

then the initial sequent to prove for this property is

$$\Gamma, \gamma, e', ty, v_1, v_2 : IH, (\gamma \vdash e' \Downarrow cons(v_1, v_2))^*, \Gamma \vdash head(e') : ty,$$
$$\forall x, ty_x, v_x . lkpTy(\Gamma, x, ty_x) \supset lkpVal(\gamma, x, v_x) \supset nilty \vdash v_x : ty_x \longrightarrow nilty \vdash v_1 : ty$$

Note that the original expression e has been replaced by $head(e')$ and the original value v has been replaced by v_1, the value to which $head(e')$ evaluates. Our restrictions on case analysis prevent us from analyzing the new evaluation derivation for e' any further, as its primary component is unstructured. We can, however, analyze the typing derivation, as its primary component is the structured $head(e')$. Doing so yields a sequent

$$\Gamma, \gamma, e', ty', v_1, v_2 : IH, (\gamma \vdash e' \Downarrow cons(v_1, v_2))^*, \Gamma \vdash e' : list(ty'),$$
$$\forall x, ty_x, v_x . lkpTy(\Gamma, x, ty_x) \supset lkpVal(\gamma, x, v_x) \supset nilty \vdash v_x : ty_x \longrightarrow nilty \vdash v_1 : ty'$$

This has the necessary premises for using the induction hypothesis, and doing so gives us a premise sequent

$$\Gamma, \gamma, e', ty', v_1, v_2 : IH, (\gamma \vdash e' \Downarrow cons(v_1, v_2))^*, \Gamma \vdash e' : list(ty'),$$
$$\forall x, ty_x, v_x . lkpTy(\Gamma, x, ty_x) \supset lkpVal(\gamma, x, v_x) \supset nilty \vdash v_x : ty_x,$$
$$nilty \vdash cons(v_1, v_2) : list(ty') \longrightarrow nilty \vdash v_1 : ty'$$

We can analyze the new typing derivation for $cons(v_1, v_2)$. This gives us new hypotheses $nilty \vdash v_1 : ty'$ and $nilty \vdash v_2 : list(ty')$. The former is exactly the conclusion we want, so we can us the *id* rule to complete the proof.

An obvious question to ask is if the restrictions on case analysis in a modular proof from Definition 4.1 make it too difficult to prove cases for interesting properties. We have found this not to be the case, for two apparent reasons. First, the induction hypothesis often obviates a second-level case analysis. Unifying the key relation with a particular rule generally unifies the primary component and some other arguments with terms built at the

top level by constructors. This also generally structures the primary components of other relations, allowing case analysis on them. In turn, thus usually gives us the premises we need for using the induction hypothesis with sub-derivations of the key relation. We see this in our example of proving a case for type preservation above, as the top-level case analysis structures the primary component of the typing derivation, allowing us to analyze it and then apply the induction hypothesis.

The second apparent reason the limitations on case analysis are not too restrictive is that we can use other properties as lemmas. In particular, when we want to use a second case analysis on a premise with an unstructured primary component argument, we can often create another property that specifies what we want. We can then prove this property in an extensible fashion, with the second case analysis we wanted as its top-level one, which is not restricted by the primary component argument being unstructured. Then we can use the lemma to show what we wanted in the original proof rather than using the disallowed case analysis.

4.3 Proofs in Introducing Modules

Recall from above that the module introducing a property is responsible for proving the property for the rules it knows and those introduced by other modules that do not know the property, the latter being instantiated default rules and independent rules. To accomplish this, we create a composed language that includes the introducing module and the modules on which it builds, as well as two modules modeling those that do not know the property. These allow us to carry out the generic reasoning required of the introducing module.

The first generic module models modules knowing the primary component category of the property's key relation, but not the property or the key relation. This module introduces a generic constructor for which the key relation's default rule will be instantiated in the composed language used for reasoning. Because the default rule will also be instantiated for constructors from the modules it models, this exactly matches the definition of the key relation for the cases it models. Because it represents modules where the key relation will

be defined by the default rule, we call it the *default rule generic module*.

The other generic module represents modules knowing the property's key relation, and, therefore, also knowing its primary component category, but not knowing the property itself. This module also introduces a generic constructor, as the modules it represents may introduce new constructors. However, the form of the rules introduced by these modules is not known, since they are introduced by the unknown modules themselves. The module introducing the key relation will introduce a *proxy rule*, a rule meant to approximate the behavior of any rule introduced by an extension to the module introducing the relation. The generic module instantiates this proxy rule for its generic constructor, thus giving it a definition of the key relation that approximates the independent rules for which it stands. We call this the *proxy rule generic module* because it uses the proxy rule to represent the rules from other extensions.

While these two generic modules give us the cases for the generic proofs, what makes them possible are projection constraints. Projection constraints are properties that define how the semantics of a term must relate to the semantics of its projection. Because default rules and proxy rules almost always use projections, projection constraints let us relate the semantics of these projections to those of the original term. This often enables creating the appropriate hypotheses for using the induction hypothesis, as well as lifting the conclusion we get from using the induction hypothesis with the projection back to the original term, the key steps in most generic proofs.

In this section, we first describe the module for creating instantiated default rules for generic reasoning. We then describe the module that uses the proxy rule to represent independent rules. After this, we discuss how the introducing module writes its proof, including the restrictions on the proof rules it uses. Finally, we discuss projection constraints and their use in writing generic proofs.

4.3.1 Default Rule Generic Module

The module introducing a property needs to prove the property holds in cases arising from the default rule for the key relation being instantiated for constructs from modules knowing

the primary component category of its key relation but not the key relation itself (*i.e.*, those labeled Y in Figure 4.1). We introduce the default rule generic module to allow it to do so. This module introduces a generic constructor to represent those introduced by modules of the type it represents. When writing a proof of the property, the introducing module will use a language composition that includes the default rule generic module. In this language, the default rule for the key relation will be instantiated for this generic constructor, giving the module's proof a generic proof case corresponding to instantiations of the default rule in any composed language. In creating the composed proof for a composed language, the proof for the case of the default rule being instantiated for the generic constructor can then be used as a proof for the case of the default rule being instantiated for a constructor from another module. This is done by replacing the generic constructor in the proof with a term built by the new constructor.

As an example of what we want from this generic module, consider Property 3.2, introduced by the optimization extension O. This property states that the optimized version of an expression evaluates if the original did, and to the same value. Because its key relation opt_e is introduced by the O extension, but the primary component category is introduced by the host language H, there can be modules knowing the primary component category but not the key relation. For example, this is the list extension's knowledge. Then the O module must prove the property will hold when the default rule for opt_e is instantiated for the list extension's constructs, but without knowing those constructs. The default rule generic module allows it to do so, introducing a generic constructor to represent them. In the language composition used for the O module to write its proof, the default rule is instantiated for this generic constructor, as it would be in a composition containing the list extension for the list extension's constructors. In creating the composed proof for a language containing both the O and L modules, each case for opt_e's default rule being instantiated for one of the list constructs is proven by taking the generic proof and replacing the generic construct with a term built by the list extension's construct.

Definition 4.2 (Default rule generic module)**.** *For a property introduced by module M with key relation R, the default rule generic module is written $\mathbb{I}(M, R)$. It builds on the modules*

on which M builds that do not also build on the module introducing R, M_R:

$$\mathbb{B}^{\mathbb{I}(M,R)} = \{N | N \in \mathbb{B}^M \wedge M_R \notin \mathbb{B}^N\}$$

This is the maximal subset of the modules on which M builds that do not know the key relation. This extension does not introduce any rules, syntax categories, or new relations. It adds one constructor, ι, that takes no arguments and builds expressions in the primary component category of R. Then we have $\mathbb{I}(M,R) = \langle \mathbb{B}^{\mathbb{I}(M,R)}, \varnothing, \{\iota\}, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing \rangle$.

We can write the default rule generic module for module O's Property 3.2. The module $\mathbb{I}(O, opt_e)$ introduces a constructor ι constructing expressions in the category e. Its builds-on set $\mathbb{B}^{\mathbb{I}(O,opt_e)}$ is $\{H\}$, as that is the only module on which O builds and H does not introduce the key relation, so we have $\mathbb{I}(O, opt_e) = \langle \{H\}, \varnothing, \{\iota\}, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing \rangle$. Because the generic module builds on H but not O, the default rule for opt_e will be instantiated for ι in a language composition including H, O, and $\mathbb{I}(O, opt_e)$, such as the composition O will use for writing its proof of Property 3.2.

4.3.2 Proxy Rule Generic Module

In addition to the cases for rules it knows and instantiated default rules, the module introducing a property also needs to prove the property holds in cases for independent rules, those introduced by extensions knowing the key relation but not the property itself (*i.e.*, those labeled Z in Figure 4.1). As for reasoning about instantiated default rules, we introduce a generic module with a generic constructor to permit it to do so, and will include this module in the language composition used for reasoning. Unlike in reasoning about instantiated default rules, we do not know the form of the rules here. Consider the security extension in proving Property 3.3 that it introduces, which has the host language's statement evaluation as its key relation. It needs to prove the property will hold for the list extension's X-SPLITLIST rule, but without knowing about the list extension's *splitlist* construct. Furthermore, the same proof must apply to any rules introduced by any other extensions. If another extension introduced a repeat-while loop, the same proof would need

to apply to the cases for its evaluation rules as applies to the case for the list extension's X-SPLITLIST rule, even though these rules would have very different forms.

To reason in such cases, we will need a way to view the semantics of such constructs without truly knowing them, one general enough to fit anything another extension might introduce. We introduce the set of proxy rules $\mathbb{Q}$, a set of rules for the relations a module introduces, for this purpose. The way to understand proxy rules is that, by introducing a proxy rule for a relation, a module is promising any actual rules introduced by extensions building on it will be subsumed by the proxy rule, which places a constraint on the behavior of extension-introduced rules. Because any actual rules from other extensions are subsumed by the proxy rule, any conclusion derived by an actual rule introduced by an extension may also be derived by the proxy rule. This subsumption of any extension-introduced rule by the proxy rule makes it possible to reason generically using the proxy rule and have the proof apply to cases for rules introduced by extension modules. We will discuss in Section 4.4.3 how extensions ensure this promise is true, trusting the promise for now.

The proxy rule set $\mathbb{Q}$ is similar to the default rule set $\mathbb{S}$ in that it gives us a view of the semantics of constructs we do not know, even though the view given by proxy rules is only an idea, not the actual definition as default rules give. The well-formedness requirements for $\mathbb{Q}$ are also similar to those for $\mathbb{S}$. The proxy rule set for a module may only introduce rules defining new relations introduced by the module, and it may only give one rule for each relation. We also require all the arguments to the conclusion of a proxy rule to be meta-variables so its form fits any conclusion an extension might introduce, even though those conclusions can use new constructs unknown to the module introducing the relation, and therefore unknown to the proxy rule. We can formalize this as a definition of well-formedness for a proxy rule set.

Definition 4.3 (Well-formed proxy rule set)**.** *A proxy rule set $\mathbb{Q}$ for a module M is well-formed if it satisfies two criteria. First, for each rule in the set, its conclusion must be of the form $R(\overline{x})$ where $\overline{x}$ is a set of unique meta-variables and R is introduced by M ($R \in \mathscr{R}^M$). Second, there is at most one rule in the set defining each relation.*

Not only are proxy rules similar to default rules in giving a view of the semantics of unknown constructs and in their well-formedness requirements, but also in the forms they often take. Just as it is common for default rules to use the projection of the conclusion's primary component, so it is also common for proxy rules to do likewise. For example, the rule introduced by the host language in $\mathbb{Q}^H$ for statement evaluation is

$$\frac{proj_s(s, s') \qquad (\gamma,\ s') \Downarrow \gamma''}{(\gamma,\ s) \Downarrow \gamma'} \text{ X-Q}$$

This rule tells us a statement's evaluation terminates whenever the statement projects and its projection's evaluation terminates, mimicking the common form of default rules of copying the definition from the projection. However, note our rule here does not *copy* the definition from the projection, instead permitting different evaluation contexts as results of the evaluations. By not fully copying the definition from the projection, the host language gives more freedom to extensions in writing their rules that must be subsumed by this one.

The generic extension we introduce for reasoning about rules from independent extensions introduces a rule that is the proxy rule instantiated for a generic constructor it also introduces. The proof the introducing module writes for the case when the key relation is derived by the instantiated proxy rule is used in creating a composed proof for the cases for rules introduced by extensions knowing the key relation but not the property, that is, independent rules. This is similar to how we use the proof for the case for the default rule instantiated for ι in the composition for other instantiations of the default rule. As there, we replace this generic extension's generic constructor with the primary component term of the actual rule in the composition, but the particular details are more complicated.

We can now define the generic module:

Definition 4.4 (Proxy rule generic module)**.** *For a property introduced by module M with key relation R, the proxy rule generic module is written $\mathbb{K}(M, R)$. It builds on the same modules as M ($\mathbb{B}^{\mathbb{K}(M,R)} = \mathbb{B}^M$). It does not introduce any new syntax categories or relations. It introduces one constructor, κ, taking no arguments and building expressions in*

the primary component category of R. It introduces one rule, which is the proxy rule r for the relation R that was introduced by M_R ($r \in \mathbb{Q}^{M_R}$) instantiated for κ ($r[\kappa/x]$ where $pc(r) = x$). Thus we have $\mathbb{K}(M, R) = \langle \mathbb{B}^M, \varnothing, \{\kappa\}, \varnothing, r[\kappa/x], \varnothing, \varnothing, \varnothing \rangle$.

In proving the security extension S's Property 3.3, which has the host language's statement evaluation as its key relation, we have a proxy rule generic module $\mathbb{K}(S, \Downarrow)$. This has the same builds-on set as the security extension itself, which is $\{H\}$, and it introduces a constructor κ building expressions in the syntax category s for statements. It also introduces one rule for κ:

$$\frac{proj_s(\kappa, s') \qquad (\gamma,\ s') \Downarrow \gamma''}{(\gamma,\ \kappa) \Downarrow \gamma'}\ \text{X-Q}(\kappa)$$

This rule instantiates the one from the proxy rule set for statement evaluation for the generic constructor κ. Then we have $\mathbb{K}(S, \Downarrow) = \langle \{H\}, \varnothing, \{\kappa\}, \varnothing, \{\text{X-Q}(\kappa)\}, \varnothing, \varnothing, \varnothing \rangle$.

4.3.3 Writing Modular Proofs

The modular proof written by a module for a property it introduces needs to prove the cases it knows, as well as those it will handle generically. Thus its modular proof will be written for the language of a module combining it and the generic extensions.

Definition 4.5 (Modular proof composition module)**.** *For a property introduced by module M with key relation R, the modular proof composition module is written $\mathbb{M}(M, R)$. This module does not introduce any new syntax categories, constructors, relations, or rules of any class. It builds on the same modules as M and M itself ($\mathbb{B}^M \cup \{M\}$), and possibly the two generic modules.*

- *If R and its primary component category are introduced by different modules, it builds on $\mathbb{I}(M, R)$ as well.*
- *If R is introduced by a module other than M, it builds on $\mathbb{K}(M, R)$ as well.*

Then we have $\mathbb{M}(M, R) = \langle \mathbb{B}^{\mathbb{M}(M,R)}, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing \rangle$.

Note we can have $\mathbb{B}^{\mathbb{M}(M,R)}$ be $\mathbb{B}^M \cup \{M\}$, $\mathbb{B}^M \cup \{M, \mathbb{I}(M,R)\}$, $\mathbb{B}^M \cup \{M, \mathbb{K}(M,R)\}$, or $\mathbb{B}^M \cup \{M, \mathbb{I}(M,R), \mathbb{K}(M,R)\}$, depending on the relationships between modules. If there can be modules knowing R's primary component category but not R itself, that is, the property fits scenarios B or D in Figure 4.1, it will include $\mathbb{I}(M,R)$ that represents such modules. If there can be modules knowing R but not building on M, that is, not knowing the property introduced by M for which we want to write a proof, fitting scenarios C or D in Figure 4.1, it will include $\mathbb{K}(M,R)$ that represents such modules. In any case, it includes M and all the modules on which M builds.

In $Lang(\mathbb{M}(M,R))$, the relation R is defined by rules known to M, those introduced by M itself and the modules on which it builds; the default rule for R instantiated for ι from the default rule generic module; and the proxy rule for R instantiated for κ from the proxy rule generic module. Then a canonical-form proof of a property using R as its key relation has cases for the known rules, the instantiated default rule, and the instantiated proxy rule arising from its top-level case analysis, or as many of these as can be relevant to a composed proof. This is one for each case which the introducing module is responsible for proving.

Recall the purpose of a module introducing a property writing a proof for $Lang(\mathbb{M}(M,R))$ is to use sub-proofs in creating a proof of the property for any composed language. A composed language will have different rules than the language for which the module's proof is written, as new rules can be added by the inclusion of other modules in a composition. To make the sub-proofs valid in a composed language, we need to ensure the language rules used in them correspond to ones that will be in a composed language where we use the sub-proofs. One way we do this is by restricting case analysis to situations where we know all the rules that may apply. As in modular proofs for imported properties (Definition 4.1), we allow case analysis if the primary component of the derivation being analyzed has a known constructor as its top-level symbol, or if the relation's primary component category is non-extensible. Because new rules introduced by extensions cannot have existing constructors as the top-level symbol of the conclusion's primary component, no rules can be added in a composition and all applicable rules are known in these cases.

The situation is a bit more complicated for the generic constructors. Consider the

generic constructor ι from the default rule generic module. The proof for the default rule instantiated for it is to be used for other constructs for which the key relation's default rule is instantiated in a composition by replacing ι with terms built by the other constructs. Then the language rules used in the proof for this case need to correspond to ones that will exist for any of the constructs for which it stands. We have ensured this for the *def*$\mathcal{R}$ proof rule by having the generic module build on all modules not knowing the key relation R. By using this maximal builds-on set, we ensure the only default rules instantiated for ι, and thus the only rules defining any relations for it, will also be instantiated for any construct for which it stands in a composition. This is because any module for which it stands also cannot build on the one introducing R, and thus the default rules from R's module and any that build on it must be instantiated for constructs for which ι stands. We can similarly limit case analysis to situations where these instantiated default rules, and any rules known to M with meta-variables as their primary component arguments, are the only ones applicable. Specifically, this means case analysis is only allowed on derivations with the generic constructor ι as the primary component argument if the relation being derived is introduced by a module not in the builds-on set of the default rule generic constructor, as its relation must be defined for such constructs by instantiating default rules.

The situation for the κ generic constructor from the proxy rule generic module is very similar to that for the ι generic constructor from the other generic module. By using the maximal set of modules not knowing the one introducing the property as the builds-on set of the generic module (*i.e.*, all the modules on which M builds), we have ensured only default rules introduced by M are instantiated for κ. These will also be instantiated for any constructor for which κ stands, as the modules introducing them must be unrelated to M. Case analysis on hypotheses with κ as the primary component argument is limited to situations where the relation being analyzed must be defined by instantiating the default rule, as for ι. This means the relation must be introduced by M itself, as this is the only module guaranteed not to be known by modules for which the proxy rule generic module stands. We must also disallow using the instantiated proxy rule with the *def*$\mathcal{R}$ proof rule. Unlike the default rules instantiated for ι and κ, which correspond exactly to rules that

will be part of a composed language for any constructs for which the generic constructors stand, this rule will not have an exact corollary. For example, the proof for the proxy rule instantiated for κ for the security extension's property will be used for the list extension's *splitlist* construct and the X-SPLITLIST rule, but X-Q(*splitlist*), the proxy rule instantiated for *splitlist*, will not be part of a composed language including both extensions. However, the default rule for the security extension's *secure* relation instantiated for *splitlist* will be part of a composed language including both extensions, as the default rule instantiated for any construct for which κ stands will be, and thus using the default rule instantiated for κ is fine.

One final consideration for the proof written by the module introducing a property is the use of the generic constructors in cases other than those for the rules defining the key relation for the generic constructors. If we used, say, ι with the $\forall\mathcal{L}$ proof rule in a case for a rule introduced by the host language in proving a property introduced by the optimization extension, a composition containing only the host language and optimization extension would not necessarily have a corresponding construct to use in the composition, one with the same rules. Thus we limit using the generic constructors to situations where we know a composed language will include corresponding constructs, those for the rules defining the key relation specifically for the generic constructs.

We can now define modular proofs written for properties by the modules introducing them, formalizing the restrictions given in the preceding paragraphs.

Definition 4.6 (Modular proof in introducing module)**.** *Let M be a module and let $P = \forall\overline{x}.R(\overline{t}) \supset F$ be a property introduced by M, constructed using the vocabulary of Lang(M). Let $\mathscr{L}$ be a set of lemmas in which ι and κ do not appear. A modular proof for P relative to its introducing module M is a canonical-form proof for P using as lemmas the properties in $\mathscr{L}$, written in the language Lang$(\mathbb{M}(M, R))$ where, in the proofs of each premise sequent of the top-level case analysis, the instantiated proxy rule introduced by $\mathbb{K}(M, R)$ is not used with the def$\mathcal{R}$ proof rule and the def$\mathcal{L}$ proof rule and its annotated variants are used to analyze hypotheses of the form $R'(\overline{t'})$ in only the following situations:*

- *the primary component argument of* $R'(\overline{t'})$ *is built by a constructor other than* ι *or* κ*,*
- *the primary component category of* R' *is a non-extensible category,*
- *the primary component argument of* $R'(\overline{t'})$ *is* ι *and* R' *is a relation introduced by a module not in* $\mathbb{B}^{\mathbb{I}(M,R)}$*, or*
- *the primary component argument of* $R'(\overline{t'})$ *is* κ *and* R' *is a relation introduced by* M *(*i.e.*,* R' *is introduced by a module not in* $\mathbb{B}^{\mathbb{K}(M,R)}$*).*

Furthermore, ι *and* κ *may only be used as the terms, or as sub-terms of the terms used, in the premises of the* $\forall\mathcal{L}$ *and* $\exists\mathcal{R}$ *rules if they are present in the conclusion sequent of the rule.*

An obvious concern is whether a module can introduce interesting properties when it needs to write a modular proof according to Definition 4.6. In Section 4.2, we argued the restrictions imposed on modular proofs for imported properties (Definition 4.1) do not make proofs difficult. The restrictions imposed on modular proofs for introducing modules include those for modular proofs of imported properties, with the addition of restrictions specific to generic reasoning. The concern, then, is whether generic reasoning, where we do not know the specifics of the constructs about which we are reasoning, allows us to prove interesting properties.

Consider the proof for the generic case for instantiated default rules for Property 3.2, that optimized expressions evaluate to the same value as their unoptimized versions. Due to the simplicity of the default rule for opt_e, the proof is immediate. Recall that the default rule OE-Default "optimizes" expressions from other modules to themselves to avoid erasing special behavior. Then the definition of opt_e for ι that creates the generic case in the modular proof is $opt_e(\iota, \iota)$. Thus we need to show $\gamma \vdash \iota \Downarrow v$, but we also have an assumption of $\gamma \vdash \iota \Downarrow v$, so we can use the *id* rule to complete the proof immediately.

This is an extremely simple case. In general, generic cases are more complex than this because the default and proxy rules defining the cases are more complex than the OE-Default rule. To prove the generic cases, we usually need some more knowledge about

the constructs for which the generic ones stand. We can get some such knowledge through allowed case analyses, such as on derivations of relations with the generic constructor as the primary component argument that must be defined by instantiating their default rules. A source of knowledge equally as important, if not more so, is the set of other properties available to use as lemmas. Since default rules and proxy rules often use projections, a particularly important subset of lemmas are those we identify as projection constraints.

4.3.4 Projection Constraints and Generic Proofs

Projection constraints are a loosely-defined subset of a language's properties defining relationships between the semantics of constructs and their projections. They constrain the ways in which extensions can define imported relations for new constructs relative to their projections. As an example of a projection constraint, our host language introduces a property requiring a statement's projection to evaluate under the same evaluation context if the statement itself evaluates:

$$\forall s, s', \gamma, \gamma'.proj_s(s, s') \supset (\gamma,\ s) \Downarrow \gamma' \supset \exists \gamma''.(\gamma,\ s') \Downarrow \gamma'' \tag{4.1}$$

The host language also introduces other projection constraints requiring the evaluation results for a statement and its projection to contain the same values for all names, and that any two projections of the same statement are equal.

Projection constraints fulfill what we noted in Section 2.1.3, that the reasoning framework lets us define what we expect of projections, and what it is important for them to preserve about the original term. By limiting how extensions define imported relations, they make it possible for other extension modules to define default rules using projections that will have the properties they want, this being formalized by writing generic proofs for either kind of generic case.

Consider the proof for the generic case representing independent rules, those introduced by modules knowing the key relation but not the property, for Property 3.3 from the security

extension. We repeat the property here for convenience:

$$\forall s, \Sigma, sl, \Sigma', \gamma_1, \gamma_1', \gamma_2, \gamma_2'.(\gamma_1,\ s) \Downarrow \gamma_1' \supset (\gamma_2,\ s) \Downarrow \gamma_2' \supset \Sigma\ sl \vdash \mathit{secure}(s, \Sigma') \supset \mathit{eqpublicvals}(\Sigma, \gamma_1, \gamma_2) \supset \mathit{eqpublicvals}(\Sigma', \gamma_1', \gamma_2')$$

In this case, we assume that the statement s is built by κ and the first evaluation is derived using X-Q(κ), so we have $(\gamma_1,\ \kappa) \Downarrow \gamma_1'$, $\mathit{proj}_s(\kappa, s')$, and $(\gamma_1,\ s') \Downarrow \gamma_1''$. We cannot analyze the derivation of evaluation under γ_2 (*i.e.*, $(\gamma_2,\ \kappa) \Downarrow \gamma_2'$) because evaluation is introduced by the host language, but we can use Projection Constraint 4.1 with it to get a derivation of evaluation for s' under γ_2, $(\gamma_2,\ s') \Downarrow \gamma_2''$. We can analyze the derivation of $\Sigma\ sl \vdash \mathit{secure}(\kappa, \Sigma')$, finding it is defined by S-Default instantiated for κ, so, as any two projections must be the same according to one of our projection constraints, we have a derivation of $\Sigma\ sl \vdash \mathit{secure}(s', \Sigma')$. The induction hypothesis shows that $\mathit{eqpublicvals}(\Sigma', \gamma_1'', \gamma_2'')$ holds. Because γ_1' and γ_1'' must have the same values for each variable due to a projection constraint introduced by the host language, and the same for γ_2' and γ_2'', we also have $\mathit{eqpublicvals}(\Sigma', \gamma_1', \gamma_2')$, our required conclusion, finishing the proof of the generic case.

This proof would not be possible without projection constraints. Without them, we would not have the premises to use the induction hypothesis, nor would we be able to lift the result back from the evaluation results for the projections to the evaluation results for the original term. It is also made possible by being able to analyze the derivation of the *secure* relation, which we know must be defined by instantiating the default rule. Thus we see that we are able to prove interesting properties when we need to write generic proofs for them, as we know new relations are defined by default rules and imported relations obey the existing properties, including especially projection constraints.

4.4 Proof Composition

In this section, we demonstrate the modular proofs specified in the previous two sections, those for a module introducing a property and those for modules importing the property, are sufficient to guarantee a proof of the property exists for *any* composed language. We do this by showing how, using these modular proofs, we can construct a composed proof of the property for a composed language.

The composed proof we will build will have the canonical form of a proof for the property. This means it has a top-level case analysis creating proof cases for the four classes of rules we identified in Section 4.1, known rules, new rules, instantiated default rules, and independent rules. We break down our discussion of building the full proof by these classes of rules.

Section 4.4.1 handles both known rules, those known to the module introducing a property, and new rules, those introduced by modules building on the one introducing a property. The key insights for these rule classes are that the modular proofs had a proof case for each rule directly, and that the restrictions on modular proofs mean the move to a composed language cannot affect the rules used in the proofs of those cases.

Section 4.4.2 shows how we can use the generic proof for the key relation's default rule instantiated for a generic constructor to prove proof cases for instantiations of the default rule in a composed language. Because of how we defined the default rule generic module, the definitions of relations for the generic constructor in the language used for writing the modular proof correspond to those for the constructs for which the default rule is instantiated in a composed language. This allows us to use the generic proof with the generic constructor replaced by an appropriate new term.

The remaining case is for independent rules, rules introduced by modules knowing the key relation but not the property. We intend to use the generic proof for the rule from the proxy rule generic module for these, but this is dependent on the proxy rule subsuming the actual rules unrelated modules will introduce. How we can show this subsumption, and how we can use it to prove the relevant cases in a composed proof, is discussed in Section 4.4.3.

Finally, Section 4.4.4 takes the pieces of the proof construction for all the rule classes

and pulls them together, showing how the full proof of any property can be constructed for any composed language from the modular proofs of it.

4.4.1 Constructing Proofs for Known and New Rule Cases

Both known rules, introduced by the module M introducing a property or a module on which it builds, and new rules, introduced by modules building on M, are known to some module that writes a modular proof. Furthermore, these modular proofs include cases coming from the top-level case analysis specific to each of these rules. Intuitively, the same rule giving rise to the proof case in the modular proof will be present in the composed language and create the same sequent from the top-level case analysis in the composed proof. Also, both Definitions 4.1 and 4.6 restrict case analysis to situations where other modules cannot add rules unifying with the premise being analyzed, so the same language rules are applicable to case analyses in the modular and composed settings. Thus we can use the exact same proof, though ranging over a different underlying language, to prove the sequent in the full proof as we did in the modular one. We formalize this in Lemma 4.9. First, we show that while we write some modular proofs in the context of $\mathit{Lang}(\mathbb{M}(M, R))$ that may contain both ι and κ, our restrictions on modular proofs prevent them from appearing in proofs of sequents in which they are not initially present, a fact we will need for lifting proofs to the composed language.

Lemma 4.7 (Non-introduction of ι)**.** *Let M be a module, R be a relation, $\mathscr{L}$ be a set of lemmas in which ι and κ do not appear, $\mathcal{S}$ be a sequent, and π be a proof of $\mathcal{S}$ in the language $\mathit{Lang}(\mathbb{M}(M, R))$ possibly using lemmas in $\mathscr{L}$ respecting the restrictions given in Definition 4.6. Then ι does not occur in any sequent in π if it does not occur in $\mathcal{S}$.*

Proof. We proceed by induction on the height of π, considering the last rule used in it. Most cases are clear. The $\forall\mathcal{R}$ and $\forall\mathcal{L}$ rules are not permitted to use terms containing ι if it is not already present in the conclusion sequent. The only cases requiring careful examination are the cases for $\mathit{def}\mathcal{R}$ and $\mathit{def}\mathcal{L}$ rules, including $\mathit{def}\mathcal{L}$ variants. For the $\mathit{def}\mathcal{R}$ rule to introduce ι into the proof would require a definition clause $\forall\overline{x}.H \triangleq B$ where the consequent of $\mathcal{S}$ is

an instance of H and B contains ι. However, the only rules in $Lang(\mathbb{M}(M,R))$ in which ι appears are instantiated default rules, so ι is also the primary component argument in H in each such rule. Then the $def\mathcal{R}$ rule cannot introduce ι.

Consider when analyzing a premise $R'(\overline{t'})$ with the $def\mathcal{L}$ rule is allowed. First, we may analyze it if its primary component is built by a constructor other than ι or κ. The only applicable rules in this case are those that are part of $Lang(M)$ and cannot contain ι. The mgu also cannot introduce ι. Next, we may analyze $R'(\overline{t'})$ if the primary component category of R' is not extensible. The only rules defining R' are then part of $Lang(M)$ and cannot contain ι. Finally, we may analyze $R'(\overline{t'})$ if its primary component is κ and R' is introduced by M. The applicable rules may be those introduced by M or the default rule for R' instantiated for κ. In either case, the rules cannot contain ι. Then ι can only be present in π if it is present in the sequent $\mathcal{S}$ it proves. ■

Lemma 4.8 (Non-introduction of κ)**.** *Let M be a module, R be a relation, $\mathscr{L}$ be a set of lemmas in which ι and κ do not appear, $\mathcal{S}$ be a sequent, and π be a proof of $\mathcal{S}$ in the language $Lang(\mathbb{M}(M,R))$ possibly using lemmas in $\mathscr{L}$ respecting the restrictions given in Definition 4.6. Then κ does not occur in π if it does not occur in $\mathcal{S}$.*

Proof. Similar to Lemma 4.7. ■

Lemma 4.9 (Lift known case proofs to composition)**.** *Let $\mathcal{S}$ be a sequent using the vocabulary of $Lang(M)$ and let π be a proof of $\mathcal{S}$ using as its set of definitions $Lang(M)$ and possibly using lemmas from $\mathscr{L}$ that contains lemmas built using the vocabulary of $Lang(C)$. Furthermore, let π be a proof obeying the restrictions given by Definition 4.1 or Definition 4.6. Let C be a module such that $M \in \mathbb{B}^C$. Then π is a proof of $\mathcal{S}$ using as its set of definitions $Lang(C)$ and possibly using lemmas from $\mathscr{L}$.*

Proof. We begin by noting the restrictions in Definition 4.1 are really a subset of those in Definition 4.6, so we can consider both cases together. Also note Lemmas 4.7 and 4.8 mean ι and κ cannot appear in any sequents in the original proof, as $\mathcal{S}$ does not contain either originally. Thus a modular proof of a sequent written using $Lang(\mathbb{M}(M,R))$ rather than

$Lang(M)$ as its set of definitions is also one using only $Lang(M)$ because it cannot use the rules arising from the generic extensions.

We proceed by induction on the height of π, and case analysis on the last rule used in the proof. For rules other than $def\mathcal{L}$ and its variants and $def\mathcal{R}$, it is clear the same proof rule will apply in both languages. Then the induction hypothesis applies to the proofs of the premise sequents.

If the last rule is the $def\mathcal{R}$ rule, the rule it used in the context of $Lang(M)$ is also present in $Lang(C)$. Then the induction hypothesis shows the premise sequent can be proven by the same proof.

The final case is that where the last rule is the $def\mathcal{L}$ rule or one of its variants. Note it does not matter here which particular version of the $def\mathcal{L}$ rule is used; any annotations do not affect the argument. Because ι and κ are not present, we have but two possibilities for the case analysis. The first possibility is that the primary component of the premise being analyzed is a constructor present in $Lang(M)$. Recall well-formed modules introducing new rules defining imported relations define those new rules so they apply to only new constructors of the primary component. Thus no new rules applicable to the case analysis could have been introduced by other modules. Similarly, instantiated default rules cannot apply, as they also have new constructors for the primary component. The other possibility is that the relation being analyzed is defined over a non-extensible type. Such a relation cannot have new rules added, as new constructors of the primary component cannot be introduced by well-formed modules. Then in both cases we have corresponding premise sequents of both case analyses, and the induction hypothesis allows us to show the premise sequents in the extended context are also proven by the same proofs as in the limited context. ■

Consider lifting the example proof we wrote in Section 4.2 for the case arising from the E-Head rule for Property 3.1 from the language of the list extension, $Lang(L)$, to the composed language containing all modules in our running example, $Lang(D)$. In this proof, we analyzed the typing derivation $\Gamma \vdash head(e') : ty$. There was one rule applicable to this,

T-Head. This rule is present and applicable in $Lang(D)$, and is the only applicable rule, and it gives us the same typing derivation $\Gamma \vdash e' : list(ty')$. Using the induction hypothesis with this to get $nilty \vdash cons(v_1, v_2) : list(ty')$ is a step that is also valid in the larger language. Finally, analyzing this new typing derivation for $cons(v_1, v_2)$ uses the T-Cons rule, and only the T-Cons rule, in both $Lang(L)$ and $Lang(D)$. In both languages, this gives us the conclusion we want, and we can use the *id* proof rule to complete the proof in both contexts.

In addition to showing the same proof will prove the same sequents in the two languages, we can also show the same sequent will appear in the proof in the setting of writing a modular proof whenever it appears in the proof for a composed language containing the module writing the modular proof.

Lemma 4.10 (Existence of modular known cases)**.** *Let M be a module and C be a module building on M. Let $\mathcal{S}$ be a sequent $\Sigma : \Gamma, R(\bar{t})^{@^i} \longrightarrow F$. If, in case analysis with the def$\mathcal{L}^{@^i}$ rule on $R(\bar{t})^{@^i}$ in the context of the language $Lang(C)$, a rule defining R introduced by a module on which M builds or by M itself unifies with $R(\bar{t})$ and creates a premise sequent, then the same rule unified with $R(\bar{t})$ in the context of $Lang(\mathbb{M}(M, R))$ and created the same premise sequent, and also in the context of $Lang(M)$.*

Proof. We have the same rule in all three settings as the set of rules is created by gathering all rules from all included modules. Then the rules have the same mgu in each setting, and thus create the same premise sequent. ■

Lemmas 4.9 and 4.10 together guarantee the cases in a full proof for known and new rules can be proven based on the modular proofs written by modules included in the language composition.

4.4.2 Constructing Proofs for Instantiated Default Rule Cases

A composed language may include instantiations of the key relation's default rule for constructs introduced by relations knowing the key relation's primary component category but not the key relation itself. These instantiations of the default rule may produce proof cases

in the top-level case analysis for our full proof. These cases correspond to the one for the default rule instantiated for the generic constructor ι from the default rule generic module. We show here how we can use the generic proof written for this rule by the module introducing the property to prove the proof case for any instantiation of the default rule in a composed language.

Clearly the two sequents, that proven for the default rule instantiated for ι in the modular proof and that for a new constructor in a composed language, differ. We start by defining the relationship between them, which requires defining the replacement of one term by another.

Definition 4.11 (Term replacement)**.** *Let c be a constructor, t be a term of the same type as c builds, and let s be another term. Replacing the constructor c with t in s, written $s[\![t/c]\!]$, is defined as replacing each sub-term of s built by the constructor c with t. We extend the definition, and its notation, to formulas, contexts, unification problems, and substitutions in the obvious manner, noting quantified variables in formulas must be renamed to avoid variable capture.*

Definition 4.12 (Instance of ι sequent)**.** *A sequent $\mathcal{S}'$ is an ι-instance of a sequent $\mathcal{S}$ as determined by a term t, written $\mathcal{S} \sim_t^\iota \mathcal{S}'$, if $\mathcal{S}$ is $\Sigma : \Gamma \longrightarrow F$ and $\mathcal{S}'$ is $\Sigma' : \Gamma[\![t/\iota]\!] \longrightarrow F[\![t/\iota]\!]$, where Σ' is Σ extended with the new variables in t.*

To illustrate this definition, consider instantiating the sequent for the generic proof case for Property 3.2 from the optimization extension for the new *cons* constructor introduced by the list extension. The initial sequent $\mathcal{S}$ for the generic case is

$$\gamma, v : (\forall e, e', \gamma, v.\mathit{opt}_e(e, e')^* \supset \gamma \vdash e \Downarrow v \supset \gamma \vdash e' \Downarrow v), \gamma \vdash \iota \Downarrow v \longrightarrow \gamma \vdash \iota \Downarrow v$$

We have another sequent $\mathcal{S}'$ to prove in the composed language:

$$\gamma, v, e_1, e_2 : (\forall e, e', \gamma, v.\mathit{opt}_e(e, e')^* \supset \gamma \vdash e \Downarrow v \supset \gamma \vdash e' \Downarrow v), \\ \gamma \vdash \mathit{cons}(e_1, e_2) \Downarrow v \longrightarrow \gamma \vdash \mathit{cons}(e_1, e_2) \Downarrow v$$

We have $\mathcal{S} \sim^{\iota}_{cons(e_1,e_2)} \mathcal{S}'$ because each occurrence of ι in $\mathcal{S}$ has been replaced by $cons(e_1, e_2)$ in $\mathcal{S}'$ and the new variables e_1 and e_2 have been added to the eigenvariable context, but nothing else has changed.

We can now show how to change a proof from one for a sequent using ι to one for a sequent replacing ι with a term built by a new constructor introduced by a module knowing the key relation's primary component category but not the key relation itself.

Lemma 4.13 (Lift generic ι proof to composition)**.** *Let M be a module and R be a relation in $Lang(M)$. Let $\mathcal{S}$ be a sequent with a proof relative to $Lang(\mathbb{M}(M, R))$ possibly using lemmas from $\mathscr{L}$ following the restrictions in Definition 4.6. Assume κ does not occur in $\mathcal{S}$ and ι and κ do not appear in $\mathscr{L}$. Let C be a module building on M. Let $t = c(\overline{y})$ be a term where $\overline{y}$ is a set of variables and c is a constructor introduced by a module N building on the module introducing the primary component of R but not building on the module introducing R itself where C builds on N. Let $\mathcal{S}'$ be a sequent such that $\mathcal{S} \sim^{\iota}_{t} \mathcal{S}'$ holds. Then there is a proof of $\mathcal{S}'$ relative to $Lang(C)$ and possibly using lemmas from $\mathscr{L}$.*

Proof. Because κ does not appear in $\mathcal{S}$, it cannot appear anywhere in the proof by Lemma 4.8, so this is maintained by each proof step.

We proceed by induction on the height of the proof of $\mathcal{S}$, considering cases for the last rule used. It is clear the same rule can apply to both $\mathcal{S}$ and $\mathcal{S}'$ for all rules other than *id* and those for definitions. For the *id* rule, any occurrences of ι in $\mathcal{S}$'s consequent and the corresponding premise have been replaced in $\mathcal{S}'$ by the same term t. Thus the *id* rule also applies to $\mathcal{S}'$.

The remaining cases are those for the *def*$\mathcal{R}$ and *def*$\mathcal{L}$ rules, and the variants of the *def*$\mathcal{L}$ rule. These cases require more careful consideration as the set of language rules has changed from $Lang(\mathbb{M}(M, R))$ to $Lang(C)$. We may have added new relations introduced by other modules and rules for them. We may also have added new rules for existing relations, but the well-formedness of modules ensures these apply to only new constructors also introduced by new modules. Finally, we have replaced default rules instantiated for ι with default rules instantiated for new constructors introduced by other modules. For

a default rule r defining a relation with x as its primary component, we had $r[\iota/x]$ in $\mathit{Lang}(\mathbb{M}(M,R))$. In $\mathit{Lang}(C)$, we instead have $r[c(\overline{y})/x]$ for each constructor c building the primary component category and introduced by a module unrelated to the module introducing the relation being defined, for variables $\overline{y}$ fresh in r. In terms of the reasoning logic $\mathcal{G}$, rather than a single definition clause $\forall\overline{x}.H[\iota/x] \triangleq B[\iota/x]$, we have a family of clauses of the form $\forall\overline{x},\overline{y}.H[c(\overline{y})/x] \triangleq B[c(\overline{y})/x]$ for various new constructors c. This final category of changed rules will be important in our proof.

Consider the case where the last proof rule in the proof of $\mathcal{S}$ is the $\mathit{def}\mathcal{R}$ rule. Then $\mathcal{S}$ must have the form $\Sigma : \Gamma \longrightarrow A$ for some atomic formula A, derived from a sequent $\Sigma' : \Gamma \longrightarrow B[\theta]$ using a definition clause $\forall\overline{x}.H \triangleq B$. Also, the sequent $\mathcal{S}'$ must have the form $\Sigma'' : \Gamma[\![t/\iota]\!] \longrightarrow A[\![t/\iota]\!]$. We can assume the domain of θ is disjoint from the variables in t; if it is not, Theorem 3.4 lets us replace it with one that is. Theorem B.1 shows that, because $A = H[\theta]$, we also have $A[\![t/\iota]\!] = H[\![t/\iota]\!][\theta[\![t/\iota]\!]]$. Consider whether ι was in the original definition clause:

- If ι was not in the original definition clause, the rule to which it corresponds is part of $\mathit{Lang}(C)$ and $H[\![t/\iota]\!] = H$, so $A[\![t/\iota]\!] = H[\theta[\![t/\iota]\!]]$. Then the $\mathit{def}\mathcal{R}$ rule applies, and we have a premise sequent $\Sigma''' : \Gamma[\![t/\iota]\!] \longrightarrow B[\theta[\![t/\iota]\!]]$. By Theorem B.1 this is equivalent to $\Sigma''' : \Gamma[\![t/\iota]\!] \longrightarrow B[\theta][\![t/\iota]\!]$, which is related to the premise sequent in the proof of $\mathcal{S}$, so the induction hypothesis applies.

- If ι was in the original definition clause, it was an instantiated default rule, and there is now an instantiation of the default rule for the constructor building t. Then $\forall\overline{x},\overline{y}.H[\![t/\iota]\!] \triangleq B[\![t/\iota]\!]$ is in $\mathit{Lang}(C)$. The sequent consequent $A[\![t/\iota]\!]$ is an instance of the head of this clause, with $\theta[\![t/\iota]\!]$ as an mgu. The premise sequent to prove for using the $\mathit{def}\mathcal{R}$ rule is $\Sigma''' : \Gamma[\![t/\iota]\!] \longrightarrow B[\![t/\iota]\!][\theta[\![t/\iota]\!]]$, which is equivalent to $\Sigma''' : \Gamma[\![t/\iota]\!] \longrightarrow B[\theta][\![t/\iota]\!]$ by Theorem B.1. This is related to the premise sequent in the proof of $\mathcal{S}$ by $\sim^{\iota}_{t}$, so the induction hypothesis applies to show we have a proof of $\mathcal{S}'$.

In either case the $\mathit{def}\mathcal{R}$ rule applies to $\mathcal{S}'$, and we have a proof of the premise sequent.

If the last rule in the proof was the *def*$\mathcal{L}$ rule or one of its annotated variants, there are three possibilities for why the case analysis was allowed on a premise $R'(\overline{t'})$. We consider each in turn.

- The primary component argument of $R'(\overline{t'})$ may have been built by a constructor other than ι. The primary component of $R'(\overline{t'})[\![t/\iota]\!]$ is still built by the same constructor. As noted above, well-formed extension modules cannot add new rules pertaining to preexisting constructs, and instantiated default rules cannot apply to preexisting constructs either. Then the only rules that might apply in the context of the composed language $Lang(C)$ are the rules from $Lang(M)$, which were the same ones considered in the original proof. Consider one of these clauses, $\forall \overline{x}.H \triangleq B$. Because it is from $Lang(M)$, ι cannot appear in it, so $H[\![t/\iota]\!]$ is the same as H. Theorem B.7 shows $\{\langle R'(\overline{t'})[\![t/\iota]\!], H[\![t/\iota]\!]\rangle\}$ has a unifier only when $\{\langle R'(\overline{t'}), H\rangle\}$ has a unifier. Then each rule unifying with $R'(\overline{t'})$ also unifies with $R'(\overline{t'})[\![t/\iota]\!]$ and vice versa. If such a rule unified, it would have an mgu θ, and a premise sequent $\Sigma[\theta] : \Gamma[\theta], B[\theta] \longrightarrow F[\theta]$. By Theorems 3.4 and B.7, we may assume $\theta[\![t/\iota]\!]$ is an mgu for $\{\langle R'(\overline{t'})[\![t/\iota]\!], H\rangle\}$. We can then choose a premise sequent $\Sigma'[\theta[\![t/\iota]\!]] : \Gamma[\![t/\iota]\!][\theta[\![t/\iota]\!]], B[\![t/\iota]\!][\theta[\![t/\iota]\!]] \longrightarrow F[\![t/\iota]\!][\theta[\![t/\iota]\!]]$ as the premise for this clause in the new proof. By Theorem B.1 this is equivalent to $\Sigma'[\theta[\![t/\iota]\!]] : \Gamma[\theta][\![t/\iota]\!], B[\theta][\![t/\iota]\!] \longrightarrow F[\theta][\![t/\iota]\!]$, which is related to the premise sequent in the original proof by $\sim_t^\iota$, and thus the induction hypothesis applies to show it can be proven. We can apply this argument to each clause, completing the proof in this case.

- The primary component category of R' may be a non-extensible category. As no new rules may be added, since no new constructors of the primary component may be added, the same argument applies as in the previous case.

- The primary component argument of $R'(\overline{t'})$ may have been ι with R' being introduced by a module not in $\mathbb{B}^{\mathbb{M}(M,R)}$. In this case, the primary component of $R'(\overline{t'})[\![t/\iota]\!]$ is now t. There are two types of rules that may be relevant here. First, there could be rules that were part of $Lang(M)$ defining R'. To be applicable, such a rule would need a

schematic variable for its primary component, as no constructor known in $Lang(M)$ could unify with t built by a constructor from an at-the-time unknown extension. These rules are also part of $Lang(C)$, and the same argument as above applies. The other possible type of rule is an instantiated default rule. Then a clause $\forall \overline{x}.H \triangleq B$ from the original language setting has a corresponding rule $\forall \overline{x}, \overline{y}.H[\![t/\iota]\!] \triangleq B[\![t/\iota]\!]$ in the new language setting. We can then use a similar argument to the one above to show this clause with ι replaced by t unifies if and only if the original rule unified, and that we have premise sequents related by $\sim_t^\iota$ if it does unify. Then the induction hypothesis applies to show the premise sequent may be proven as well.

Then there is a proof for the new sequent in the new language setting regardless of which case we have for allowed use of the *def*$\mathcal{L}$ proof rule, and thus for any proof rule. ■

Lemma 4.13 proves that if we have a proof of a sequent for the default rule instantiated for ι in the modular proof written by the module introducing the property, we can use it to build a proof of a sequent for an instantiation of the default rule in the composed language. However, this is only useful if we know that such a proof will exist when we need it for a composition, which we show in Lemma 4.14.

Lemma 4.14 (Existence of generic ι case)**.** *Let M be a module and C be a module building on M. Let $\mathcal{S}$ be a sequent $\Sigma : \Gamma, R(\overline{t})^{@^i} \longrightarrow F$. If, in case analysis with the def$\mathcal{L}^{@^i}$ rule on $R(\overline{t})^{@^i}$ in the context of $Lang(C)$, the default rule for R instantiated for a constructor c from a module unrelated to M unifies with $R(\overline{t})^{@^i}$ and creates a premise sequent, then $Lang(\mathbb{M}(M, R))$ contains a generic constructor ι and the default rule for R instantiated for it, and case analysis with the def$\mathcal{L}^{@^i}$ rule on $R(\overline{t})^{@^i}$ in the context of $Lang(\mathbb{M}(M, R))$ has this instantiated default rule unify with $R(\overline{t})^{@^i}$ and creates a premise sequent. Furthermore, the two premise sequents are related by $\sim_{c(\overline{y})}^\iota$ for appropriate variables $\overline{y}$.*

Proof. The existence of a constructor from another module for which the default rule is instantiated means the relation R and its primary component category were introduced in separate modules. Then $\mathbb{I}(M, R)$ is included in the builds-on set for $\mathbb{M}(M, R)$ and the

default rule for R is instantiated for ι because c builds R's primary component type and the modules introducing R and ι are unrelated.

Let $\forall \overline{x}.H \triangleq B$ be the clause for the default rule instantiated for ι. Then the default rule instantiated for c is $\forall \overline{x}, \overline{y}.H[\![c(\overline{y})/\iota]\!] \triangleq B[\![c(\overline{y})/\iota]\!]$ where $\overline{y}$ is appropriate variables fresh to the rule. By Theorems B.6 and B.7, $\{\langle R(\overline{t}), H\rangle\}$ has an mgu if and only if $\{\langle R(\overline{t}), H[\![c(\overline{y})/\iota]\!]\rangle\}$ has an mgu. Furthermore, by Theorem B.6, we can assume the mgu θ for the problem containing ι is such that $\theta[\![c(\overline{y})/\iota]\!]$ is an mgu for the problem replacing ι with $c(\overline{y})$. Then the premise sequents are $\Sigma[\theta] : \Gamma[\theta], B[\theta]^{*^i} \longrightarrow F[\theta]$ and $\Sigma[\theta[\![c(\overline{y})/\iota]\!]] : \Gamma[\theta[\![c(\overline{y})/\iota]\!]], B[\theta[\![c(\overline{y})/\iota]\!]]^{*^i} \longrightarrow F[\theta[\![c(\overline{y})/\iota]\!]]$. By Theorem B.1, we can rewrite the second sequent as $\Sigma[\theta[\![c(\overline{y})/\iota]\!]] : \Gamma[\theta][\![c(\overline{y})/\iota]\!], B[\theta][\![c(\overline{y})/\iota]\!] \longrightarrow F[\theta][\![c(\overline{y})/\iota]\!]$. It is then clear the two premise sequents are related by $\sim^{\iota}_{c(\overline{y})}$. ■

With Lemmas 4.13 and 4.14, we can show that any cases for instantiations of the default rule for the key relation in a full proof of a property with canonical form are provable in the context of the full language.

4.4.3 Constructing Proofs for Independent Rule Cases

We have shown thus far how the modular proofs written in limited contexts can prove the corresponding cases in a proof for a composed language, both for rules directly introduced by some module knowing the property and thus present in a composed language and for instantiations of default rules. However, this is more difficult for independent rules, rules written for a property's key relation without knowledge of the property. We propose the proxy rule generic module as the key to these proofs, with the case for the rule it introduces, which defines the key relation for κ, as a modifiable proof for the new cases. This is similar to how the case for the key relation's default rule instantiated for ι is a modifiable proof for all instantiations of the default rule.

The ways we use the two cases are a bit different due to how the rules defining the generic cases relate to those present in the composition. The two instantiations of the default rule, the modular one for the generic constructor ι and one in the composed language, are related

simply by replacing the generic constructor with a term built by a new constructor. In the situation we are currently considering, the rules are not related in this way. The proxy rule defining the key relation in $\mathbb{Q}$ used to construct the proxy rule generic module $\mathbb{K}(M, R)$ is generally not related to those other extensions will introduce by direct term replacement.

To close the gap, we introduce the *proxy version of a relation*. The proxy version of a relation will incorporate both the true definition of the relation and the proxy rule for it. By using the proxy version of the key relation in the place of the key relation, we can use the generic proof case from the modular proof, which assumed the proxy rule defined the relation, to show the property holds for the true definition. To do this, however, we must show the proxy version of the key relation is a true model of the key relation, that it fulfills the promise of the module introducing the proxy rule that the proxy rule subsumes all extension-introduced rules. In the rest of this subsection, we formally define the proxy version of a relation, describe how we use it to build proofs for composed languages from modular proofs, and discuss showing its equivalence to the original relation.

Proxy Version of a Relation

The proxy version of a relation incorporates the proxy rule from $\mathbb{Q}$ into rules introduced by modules extending the one introducing the relation. It does this by adding the proxy rule's premises to the extension-introduced rules.

Definition 4.15 (Proxy version of a relation)**.** *Let M be a module introducing a relation R that is not defined mutually-recursively with any other relation and a proxy rule for R in $\mathbb{Q}^M$. Let C be either M itself or a module building on M. The proxy version of R, written R_P, is defined for Lang(C) by making a modified version of each rule in Lang(C) defining R. There are two classes of rules:*

- *For a rule in $\mathbb{R}^M$ or a default rule in $\mathbb{S}^M$ of the form*

$$\frac{\overline{R(\overline{s})} \qquad \overline{U}}{R(\overline{t})}$$

where $\overline{R(\overline{s})}$ *denotes a set of premises built by* R *and* $\overline{U}$ *denotes a set of premises of other forms, we have a rule defining* R_P *as*

$$\frac{\overline{R_P(\overline{s})} \qquad \overline{U}}{R_P(\overline{t})}$$

*in the same rule set (*e.g.*,* $\mathbb{R}^M$*). This is an identical rule other than replacing* R *with* R_P*.*

- *Suppose we have a rule in* $\mathbb{R}^N$ *for a module* N *building on* M *of the form*

$$\frac{\overline{R(\overline{s})} \qquad \overline{U}}{R(t_1, \ldots, t_n)}$$

where $\overline{R(\overline{s})}$ *denotes a set of premises built by* R *and* $\overline{U}$ *denotes a set of premises of other forms. Suppose we also have a rule in* $\mathbb{Q}^M$ *of the form*

$$\frac{\overline{R(\overline{v})} \qquad \overline{W}}{R(x_1, \ldots, x_n)}$$

where each x_i *is a distinct variable,* $\overline{R(\overline{v})}$ *is a set of premises built by* R*, and* $\overline{W}$ *is a set of premises of other forms. We have a rule*

$$\frac{\overline{R_P(\overline{s})} \qquad \overline{U} \qquad \overline{R_P(\overline{v})[\theta]} \qquad \overline{W[\theta]}}{R_P(t_1, \ldots, t_n)}$$

in $\mathbb{R}^N$ *defining* R_P *where* θ *is the substitution* $\{\langle x_1, t_1\rangle, \ldots, \langle x_n, t_n\rangle\}$*. This rule includes both the premises of the rule defining* R *and those of the proxy rule, but appropriately instantiated for the new conclusion.*

These are the only rules in the definition of R_P*.*

Figure 4.2 shows a few of the rules for statement evaluation and the corresponding

$$\frac{(\gamma,\ s_1) \Downarrow \gamma' \qquad (\gamma',\ s_2) \Downarrow \gamma''}{(\gamma,\ seq(s_1, s_2)) \Downarrow \gamma''}\ \text{X-Seq}$$

$$\frac{(\gamma,\ s_1)\ \Downarrow_P\ \gamma' \qquad (\gamma',\ s_2)\ \Downarrow_P\ \gamma''}{(\gamma,\ seq(s_1, s_2))\ \Downarrow_P\ \gamma''}\ X_P\text{-Seq}$$

$$\frac{\begin{array}{c}\gamma \vdash e \Downarrow cons(v_1, v_2) \qquad n_{hd} \neq n_{tl} \qquad update(\gamma, n_{hd}, v_1, \gamma') \\ update(\gamma', n_{tl}, v_2, \gamma'')\end{array}}{(\gamma,\ splitlist(n_{hd}, n_{tl}, e)) \Downarrow \gamma''}\ \text{X-Splitlist}$$

$$\frac{\begin{array}{c}\gamma \vdash e \Downarrow cons(v_1, v_2) \qquad n_{hd} \neq n_{tl} \qquad update(\gamma, n_{hd}, v_1, \gamma') \\ update(\gamma', n_{tl}, v_2, \gamma'') \qquad proj_s(splitlist(n_{hd}, n_{tl}, e), s') \qquad (\gamma,\ s')\ \Downarrow_P\ \gamma'''\end{array}}{(\gamma,\ splitlist(n_{hd}, n_{tl}, e))\ \Downarrow_P\ \gamma''}\ X_P\text{-Splitlist}$$

$$\frac{\gamma \vdash e \Downarrow v}{(\gamma,\ secdecl(n, ty, sl, e)) \Downarrow consval(n, v, \gamma)}\ \text{X-Secdecl}$$

$$\frac{\gamma \vdash e \Downarrow v \qquad proj_s(secdecl(n, ty, sl, e), s') \qquad (\gamma,\ s')\ \Downarrow_P\ \gamma'}{(\gamma,\ secdecl(n, ty, sl, e))\ \Downarrow_P\ consval(n, v, \gamma)}\ X_P\text{-Secdecl}$$

Figure 4.2: Example rules for statement evaluation and its proxy version

rules in the proxy version of it. The X-Seq rule introduced by the host language has the corresponding X_P-Seq rule for the proxy version. This does not add any premises as the original rule is introduced by the module introducing the relation, only changing the existing premises to use the proxy version. The other two rules shown for the proxy version of evaluation, X_P-Splitlist and X_P-Secdecl, keep their existing premises but add those from the proxy rule for statement evaluation, the X-Q rule from Section 4.3.2. X_P-Splitlist adds premises for a projection of the *splitlist* and evaluation of the projection, while X_P-Secdecl adds the same premises but projecting the *secdecl* term.

We use the proxy version of the key relation of a property in its place in building a full proof for a composed language. The restriction limiting the proxy version to relations not defined mutually-recursively with others will ensure the key relation remains the proxy version as we work our way through composed proofs, as we shall see in Lemma 4.17 below. Essentially, if a relation is defined mutually-recursively with another, they are in spirit two parts of the same relation. Then creating the proxy version of one of the mutually-recursive

relations is not truly creating the proxy version of the full relation, and the modular proof's reasoning in terms of the relation itself is reasoning about a relation with more rules than the proxy version of it.

Additionally, in order to use the proxy version in place of the key relation itself, we need two properties about it. The first we call the $dropP(R)$ property, that any derivation of the proxy version of a relation can be used to produce an equivalent one of the relation itself:

$$\forall \overline{x}.R_P(\overline{x}) \supset R(\overline{x})$$

This is clearly true, as both classes of rules in the definition of the proxy version of the relation contain premises corresponding to those for a rule defining the relation itself. A proof of it may be constructed mechanically. The second property we call the $addP(R)$ property, going the opposite direction of $dropP(R)$. This has the form

$$\forall \overline{x}.R(\overline{x}) \supset F_R \supset R_P(\overline{x})$$

where F_R is a formula specific to proving $addP(R)$ for a particular relation R. We will discuss the role of F_R in proving $addP(R)$ below, for now accepting that it is important to include. The $addP(R)$ property, in contrast to the $dropP(R)$ property, requires an explicit proof. We will assume it is true in describing how we use the proxy version of a relation to prove properties, then return to how we can prove it holds after seeing its use.

Lifting Proofs to the Proxy Version of a Relation

When we want to build a composed proof of a property of the form $\forall \overline{x}.R(\overline{t}) \supset F$ where unrelated modules may introduce new rules defining the key relation R, that is, a property fitting scenario C or scenario D in Figure 4.1, we will do so by proving another property using the proxy version of the key relation. Specifically, we will use the modular proofs to construct a proof of $\forall \overline{x}.R_P(\overline{t}) \supset F$, then use the $addP(R)$ property with this to prove the property we want about the relation R. By using the proxy version of the key relation in place of the key relation itself, we ensure the cases for independent rules in the composition

have the hypotheses of the proxy rule used in writing the generic proof, as the rules for the proxy version of the key relation include them.

To build a proof of the modified property from the modular proofs, we will need to transform proofs over the key relation itself into proofs over the proxy version of it. We define a relation on sequents to relate sequents where certain occurrences of R, specifically those with annotations $@^i$ or $*^i$ for some i, are replaced by R_P.

Definition 4.16 (Proxy version of a sequent)**.** *Let $\mathcal{S} = \Sigma : \Gamma \longrightarrow F$ and $\mathcal{S}' = \Sigma' : \Gamma' \longrightarrow F'$ be sequents. We say $\mathcal{S}'$ is an R-i-proxy version of $\mathcal{S}$, written $\mathcal{S} \sqsubseteq^i_R \mathcal{S}'$, if there is a subset Γ'' of Γ' such that each formula in Γ maps to one in Γ'' and F maps to F' by replacing each formula of the form $R(\overline{t})^{@^i}$ with $R_P(\overline{t})^{@^i}$ and each formula of the form $R(\overline{t})^{*^i}$ with $R_P(\overline{t})^{*^i}$.*

In the security module's modular proof of its Property 3.3, we have a case for the E-SEQ rule. Its initial sequent $\mathcal{S}$ is

$$
\begin{array}{l}
s_1, s_2, \Sigma, sl, \Sigma', \gamma_1, \gamma_1', \gamma_1'', \gamma_2, \gamma_2' : \\
\quad (\forall s, \Sigma, sl, \Sigma', \gamma_1, \gamma_1', \gamma_2, \gamma_2'.((\gamma_1,\ s) \Downarrow \gamma_1')^* \supset (\gamma_2,\ s) \Downarrow \gamma_2' \supset \Sigma\ sl \vdash \mathit{secure}(s, \Sigma') \supset \\
\qquad \mathit{eqpublicvals}(\Sigma, \gamma_1, \gamma_2) \supset \mathit{eqpublicvals}(\Sigma', \gamma_1', \gamma_2')), \\
\qquad ((\gamma_1,\ s_1) \Downarrow \gamma_1'')^*, ((\gamma_1'',\ s_2) \Downarrow \gamma_1')^*, (\gamma_2,\ \mathit{seq}(s_1, s_2)) \Downarrow \gamma_2', \\
\quad \Sigma\ sl \vdash \mathit{secure}(\mathit{seq}(s_1, s_2), \Sigma'), \mathit{eqpublicvals}(\Sigma, \gamma_1, \gamma_2) \longrightarrow \mathit{eqpublicvals}(\Sigma, \gamma_1', \gamma_2')
\end{array}
$$

We have a corresponding sequent $\mathcal{S}'$ that replaces each evaluation derivation with a $*$ annotation with its proxy version:

$$
\begin{array}{l}
s_1, s_2, \Sigma, sl, \Sigma', \gamma_1, \gamma_1', \gamma_1'', \gamma_2, \gamma_2' : \\
\quad (\forall s, \Sigma, sl, \Sigma', \gamma_1, \gamma_1', \gamma_2, \gamma_2'.((\gamma_1,\ s)\ \Downarrow_P\ \gamma_1')^* \supset (\gamma_2,\ s) \Downarrow \gamma_2' \supset \Sigma\ sl \vdash \mathit{secure}(s, \Sigma') \supset \\
\qquad \mathit{eqpublicvals}(\Sigma, \gamma_1, \gamma_2) \supset \mathit{eqpublicvals}(\Sigma', \gamma_1', \gamma_2')), \\
\qquad ((\gamma_1,\ s_1)\ \Downarrow_P\ \gamma_1'')^*, ((\gamma_1'',\ s_2)\ \Downarrow_P\ \gamma_1')^*, (\gamma_2,\ \mathit{seq}(s_1, s_2)) \Downarrow \gamma_2', \\
\quad \Sigma\ sl \vdash \mathit{secure}(\mathit{seq}(s_1, s_2), \Sigma'), \mathit{eqpublicvals}(\Sigma, \gamma_1, \gamma_2) \longrightarrow \mathit{eqpublicvals}(\Sigma, \gamma_1', \gamma_2')
\end{array}
$$

Note that this leaves the unannotated derivations of evaluation alone, not changing them to the proxy version. Then we have $\mathcal{S} \sqsubseteq^1_{\Downarrow} \mathcal{S}'$.

We can now show we can build a proof of the proxy version of a sequent if we have a proof of the original sequent, both proofs being for the same language.

Lemma 4.17 (Lift proof to proxy version of a sequent)**.** *Let M be a module and $\mathcal{S}$ be a sequent. Let $\mathcal{S}'$ be a sequent such that $\mathcal{S} \sqsubseteq^i_R \mathcal{S}'$ holds. Assume $\mathcal{S}$ has a proof in Lang(M) possibly using lemmas from the set of lemmas $\mathscr{L}$ including dropP(R). Then $\mathcal{S}'$ also has a proof in Lang(M) possibly using lemmas from the set of lemmas $\mathscr{L}$.*

Proof. We proceed by induction on the height of the original proof, considering cases on the last rule used in the proof. The cases for rules other than those using definitions or the *id* rule and its variants are simple. The related formulas in the two sequents have the same form, so the same proof rule applies in $\mathcal{S}'$ as in $\mathcal{S}$, and the induction hypothesis applies to the premise sequents.

If the last rule used in the proof was the *id* rule or one of its variants, consider which one. If it was the basic *id* rule, neither the consequent nor the hypothesis with which it is used can have been replaced with the proxy version of R, so the *id* rule still applies. The same is true if it was a version of the *id* rule with a different annotation number than i. If the last rule was id^{**}, $id^{@@}$, or $id^{*@}$, both the consequent and the hypothesis have replaced R with R_P, and the same version of the *id* rule applies. Finally, suppose the last rule was id^{*} or $id^{@}$. Then the hypothesis has replaced R with R_P, but the consequent has not. We can apply the *dropP*(R) lemma to the premise in $\mathcal{S}'$, then apply the *id* rule to complete the proof.

If the last rule in the original proof was the *def*$\mathcal{R}$ rule, note the consequent of $\mathcal{S}$ must be atomic and cannot have been of the forms $R(\bar{t})^{@^i}$ or $R(\bar{t})^{*^i}$. Then the consequent of $\mathcal{S}'$ is the same as that of $\mathcal{S}$, and the same definition clause can be applied. The induction hypothesis then shows the premise sequent can also be proven.

Finally, suppose the last rule was the *def*$\mathcal{L}$ rule or one of its annotated variants. If it was the basic *def*$\mathcal{L}$ rule or an annotated variant with a different annotation number, the

same language rules apply, creating related premise sequents, and the induction hypothesis can be used to show they can also be proven. Suppose the last rule was the $def\mathcal{L}^{*^i}$ rule. Then the premise being analyzed in the proof for $\mathcal{S}$ had the form $R(\bar{t})^{*^i}$ or $Q(\bar{t})^{*^i}$ for some relation Q. If it had the form $R(\bar{t})^{*^i}$, then the corresponding hypothesis in $\mathcal{S}'$ has the form $R_P(\bar{t})^{*^i}$. The rules defining the proxy version of R have the same conclusions, modulo replacing R with R_P, as those for R. Therefore corresponding rules for R and R_P either both unify or fail to unify with $R(\bar{t})^{*^i}$ and $R_P(\bar{t})^{*^i}$. Furthermore, those that unify have the same mgu and produce premise sequents related by $\sqsubseteq^i_R$, since the rules for R_P have premises related to those for the corresponding rules for R, only possibly more premises. Then the induction hypothesis applies to the premise sequents to finish the proof. If the analyzed premise had the form $Q(\bar{t})^{*^i}$, the same hypothesis is in $\mathcal{S}'$. Note no rule defining Q can have premises using R, as Definition 4.15 requires R not to be defined mutually-recursively with any other relation. Then a similar argument applies for the new premise sequents from the case analysis in the composition being related by $\sqsubseteq^i_R$ to the sequents in the original proof, and the induction hypothesis applies to the premise sequents to finish the proof. The case for when the last proof rule is $def\mathcal{L}^{@^i}$ is similar, but only requires considering the premise being analyzed being constructed by the key relation R. ■

Lemma 4.17 will permit us to lift proofs for cases for known rules, new rules, and instantiated default rules into proofs for related sequents using the proxy version of the key relation instead. In building the proof of the modified property using the proxy version of the key relation, we will use Lemmas 4.9 and 4.13 to lift the proofs to the full composed language first, then Lemma 4.17 to lift them to the proxy version.

We will take a slightly different approach for cases for independent rules, that is, rules represented by the rule from the proxy rule generic module. We will first lift the generic proof to the projection relation in $Lang(\mathbb{M}(M, R))$, then lift that proof to the composed language for the actual rule for R_P present in the composition. Lemma 4.18 specializes Lemma 4.17 to accomplish the first step.

Lemma 4.18 (Lift proof to proxy version of a sequent with restrictions)**.** *Let M be a*

module and $\mathcal{S}$ be a sequent. Let $\mathcal{S}'$ be a sequent such that $\mathcal{S} \sqsubseteq^i_R \mathcal{S}'$ holds. Assume $\mathcal{S}$ has a proof in Lang(M) possibly using lemmas from the set of lemmas $\mathscr{L}$ including dropP(R) where the proof obeys the restrictions from Definition 4.6. Then $\mathcal{S}'$ also has a proof in Lang(M) possibly using lemmas from the set of lemmas $\mathscr{L}$ obeying the restrictions from Definition 4.6.

Proof. The proof is similar to that of Lemma 4.17. Note that, in building the proof of that lemma in the case for the *def*$\mathcal{L}$ rule and its variants, the case analysis is either on a premise with exactly the same form, or one where a relation has been exchanged for its proxy version. In either case, the restrictions, which depend on where a relation was introduced and its primary component argument, are still obeyed. ■

The above lemma takes a proof of a sequent for the generic case from the proxy rule generic module for a key relation R and lifts it to one for the proxy version of this relation while remaining in the same language *Lang*$(\mathbb{M}(M, R))$. The next step is to eliminate the generic constructor, replacing it with a term built by a constructor from a module represented by the proxy rule generic extension. We define this replacement for a sequent, which is very similar to replacing ι as defined in Definition 4.12, then prove we can build a proof for the replaced sequent in a composed language from the proof we get from Lemma 4.18.

Definition 4.19 (Instance of κ sequent)**.** *A sequent $\mathcal{S}'$ is a κ-instance of a sequent $\mathcal{S}$ as determined by a term t, written $\mathcal{S} \sim^\kappa_t \mathcal{S}'$, if $\mathcal{S}$ is $\Sigma : \Gamma \longrightarrow F$ and $\mathcal{S}'$ is $\Sigma' : \Gamma[\![t/\kappa]\!] \longrightarrow F[\![t/\kappa]\!]$, where Σ' is Σ extended with the new variables in t.*

Lemma 4.20 (Lift generic κ proof to new constructors in composition)**.** *Let M be a module and R be a relation in Lang(M). Let $\mathcal{S}$ be a sequent with a proof relative to Lang$(\mathbb{M}(M, R))$ possibly using lemmas from $\mathscr{L}$ following the restrictions in Definition 4.6, and assume ι does not occur in $\mathcal{S}$. Let C be a module building on M. Let $t = c(\overline{y})$ be a term where $\overline{y}$ is a set of variables and c is a constructor introduced by a module N building on the module introducing the key relation R but not building on the module M itself where C builds on N. Let $\mathcal{S}'$ be a sequent such that $\mathcal{S} \sim^\kappa_t \mathcal{S}'$ holds. Then there is a proof of $\mathcal{S}'$ relative to Lang(C) and possibly using lemmas from $\mathscr{L}$.*

Proof. Note the similarities between Definition 4.19 and Definition 4.12. Due to these similarities, the proof here is also very similar to the proof of Lemma 4.13; in fact, the two proofs are nearly identical. Therefore we rely on the proof of that lemma for most of the details, mentioning here only the relevant changes for case analysis due to the presence of κ rather than ι.

If the last rule in the proof of $\mathcal{S}$ is the *def*$\mathcal{R}$ rule, consider whether κ was in the definition clause used. If it was not, the same rule is also part of $\mathit{Lang}(C)$, and the same argument from Lemma 4.13 applies. If κ was in the clause, the ban on using the rule from $\mathbb{K}(M, R)$ with the *def*$\mathcal{R}$ rule means it must be a default rule instantiated for κ, and this default rule must be from the default rule set $\mathbb{S}^M$. As the module introducing the constructor c that is the top-level constructor of t is not related to M, the same default rule is instantiated for c in $\mathit{Lang}(C)$, and the argument from Lemma 4.13 applies.

If the last rule in the proof was the *def*$\mathcal{L}$ rule or one of its annotated variants, there are three possibilities for why the case analysis on $R'(\overline{t'})$ was allowed. If the primary component of $R'(\overline{t'})$ was built by a constructor other than κ, or if the primary component category of R' is non-extensible, the argument for the same cases from Lemma 4.13 applies. The case analysis may also have been allowed because the primary component argument of $R'(\overline{t'})$ is κ and R' is a relation introduced by M. As in Lemma 4.13, the relevant rules are rules in $\mathit{Lang}(M)$ and the default rule instantiated for κ. Then the same argument from Lemma 4.13 applies. ■

Using Lemmas 4.18 and 4.20, we can lift the proof of the generic case from the modular proof written by the module introducing a property into one for a sequent using R_P in the context of a composed language, where the primary component, formerly the generic κ, is built by a constructor from another module with arguments that are variables fresh to the sequent. The last change we will need to use the modified proof for the case for the rule in the composition is to fill in the other arguments for the relation, and possibly other arguments to the top-level constructor of the term with which we replaced the generic construct. Recall the rules for R_P are built from those for R, and the rules for R may have any terms as

arguments, including the constructor having other terms as its arguments. For example, the X_P-SECDECL rule has the conclusion $(\gamma,\ secdecl(n, ty, sl, e)) \Downarrow_P consval(n, v, \gamma)$, but our lemmas so far have only lifted the generic proof to $(\gamma,\ secdecl(n, ty, sl, e)) \Downarrow_P \gamma'$. To complete the process, we can use Theorem 3.4 to build a proof of the sequent with a substitution filling in the expected terms for the arguments, completing the transformation of the proof written as part of the modular proof into one for the sequent we will need to prove in a composed proof. We formalize the complete transformation in Lemma 4.21, along with showing there will be an original proof to lift.

Lemma 4.21 (Existence and lifting of generic κ cases to composition)**.** *Let M be a module, C be a module building on M, and N be a module unrelated to M but on which C builds. That is, we have $M \in \mathbb{B}^C$, $N \in \mathbb{B}^C$, $M \notin \mathbb{B}^N$, and $N \notin \mathbb{B}^M$. Assume the module introducing R introduces a proxy rule for it. Let $\mathcal{S}'$ be a sequent $\Sigma' : \Gamma', R_P(\bar{t})^{@^i} \longrightarrow F$ in the language of Lang(M) and $\mathcal{S}$ be a sequent $\Sigma : \Gamma, R(\bar{t})^{@^i} \longrightarrow F$ such that $\mathcal{S} \sqsubseteq^i_R \mathcal{S}'$ holds. If, in case analysis with the def$\mathcal{L}^{@^i}$ rule on $R_P(\bar{t})^{@^i}$ in $\mathcal{S}'$ in the context of the language Lang(C), a rule defining R_P corresponding to a rule defining R introduced by module N unifies with $R_P(\bar{t})$ and creates a premise sequent $\mathcal{S}''$, then*

1. *$\mathbb{K}(M, R)$ is in $\mathbb{B}^{\mathbb{M}(M,R)}$ and its rule unifies with $R(\bar{t})$ in case analysis with the def$\mathcal{L}^{@^i}$ proof rule in the context of Lang$(\mathbb{M}(M, R))$ for $\mathcal{S}$ creating a premise sequent $\mathcal{S}_0$*
2. *and if there is a proof π of $\mathcal{S}_0$ in the context of the language Lang$(\mathbb{M}(M, R))$ and possibly using lemmas in the set $\mathscr{L}$, where each lemma uses the vocabulary of Lang(M) and where π obeys the restrictions from Definition 4.6, there is a proof of $\mathcal{S}''$ in the context of Lang(C) possibly using lemmas in $\mathscr{L}$.*

Proof. Assume the relation R was introduced by a module M_R on which both M and N built. This must be the module structure in order for N to introduce a rule for R. Then $\mathbb{K}(M, R)$ is included in the builds on set of $\mathbb{M}(M, R)$, and it introduces a rule defining R for κ, the instantiation of the proxy rule for R.

To show the rule from $\mathbb{K}(M, R)$ unifies, we work backward from the unification of the rule from N. This rule has as its conclusion $R_P(s_1, \ldots, s_m)$. The primary component argument

s_j is built by a constructor c introduced by N, since N is a well-formed module that does not introduce R. Since $R_P(\bar{t})$ is built by the vocabulary of $\mathit{Lang}(M)$, the constructor c does not occur in it. Then, since $R_P(\bar{t}) = R(t_1, \ldots, t_m)$ and $R_P(s_1, \ldots, s_m)$ unify, t_j unifies with s_j, and thus t_j must be a meta-variable. Since the conclusion of the rule from $\mathbb{K}(M, R)$ must have the form $R(x_1, \ldots, x_{j-1}, \kappa, x_{j+1}, \ldots, x_m)$ where each x_k is a distinct variable (this is a requirement for a well-formed proxy rule set $\mathbb{Q}$), this unifies with $R(\bar{t})$. Then we have a premise sequent $\mathcal{S}_0$ for the case analysis in $\mathcal{S}$. This completes the proof of the first part.

The clause in $\mathit{Lang}(\mathbb{M}(M, R))$ for the rule from $\mathbb{Q}^{M_R}$ instantiated for κ has the form

$$\forall x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_m. R(x_1, \ldots, x_{j-1}, \kappa, x_{j+1}, \ldots, x_m) \triangleq \exists \overline{y}. \overline{U} \wedge \overline{R(\overline{q})}$$

where $\overline{U}$ is premises not built by R. One mgu for unifying this with $R(\bar{t})$ is

$$\phi = \{\langle x_1, t_1\rangle, \ldots, \langle x_{j-1}, t_{j-1}\rangle, \langle t_j, \kappa\rangle, \langle x_{j+1}, t_{j+1}\rangle, \ldots, \langle x_m, t_m\rangle\}$$

We can assume π and $\mathcal{S}_0$ use this mgu; if they don't, Theorem 3.4 lets us produce a sequent and proof that do. Then $\mathcal{S}_0$ is $\Sigma[\phi] : \Gamma[\phi], (\exists \overline{y}. \overline{U} \wedge \overline{R(\overline{q})^{\mathbb{Q}^i}})[\phi] \longrightarrow F[\phi]$. Using Lemma 4.18, we can get a proof of a sequent $\mathcal{S}_1 = \Sigma'[\phi] : \Gamma'[\phi], (\exists \overline{y}. \overline{U} \wedge \overline{R_P(\overline{q})^{\mathbb{Q}^i}})[\phi] \longrightarrow F[\phi]$ in the context of $\mathit{Lang}(\mathbb{M}(M, R))$ and possibly using lemmas from $\mathscr{L}$. Note we now have Σ' and Γ' from $\mathcal{S}'$ instead of Σ and Γ from $\mathcal{S}$. We have assumed the new variables from ϕ are distinct from those in Γ'. Because $\mathcal{S}$ and $\mathcal{S}'$ were originally related by $\sqsubseteq^i_R$, $\mathcal{S}_0$ and $\mathcal{S}_1$ are now related by $\sqsubseteq^i_R$.

Let $\mathcal{S}_2$ be a sequent $\Sigma[\phi], \overline{z} : \Gamma'[\phi][\![c(\overline{z})/\kappa]\!], (\exists \overline{y}. \overline{U} \wedge \overline{R_P(\overline{q})^{\mathbb{Q}^i}})[\phi][\![c(\overline{z})/\kappa]\!] \longrightarrow F[\phi][\![c(\overline{z})/\kappa]\!]$ where $\overline{z}$ is a set of variables fresh to the sequent. Thus we have $\mathcal{S}_1 \sim^{\kappa}_{c(\overline{z})} \mathcal{S}_2$. Lemma 4.20 gives us a proof of $\mathcal{S}_2$ in the context of $\mathit{Lang}(C)$ and possibly using lemmas from $\mathscr{L}$. Note that, because our original sequents $\mathcal{S}$ and $\mathcal{S}'$ are built by the vocabulary of $\mathit{Lang}(M)$, of which κ is not part, the only appearances of κ in the sequent before replacing κ are from ϕ. Then $\mathcal{S}_2$ can be rewritten using $\phi' = \phi[\![c(\overline{z})/\kappa]\!]$, that is,

$$\phi' = \{\langle x_1, t_1\rangle, \ldots, \langle x_{j-1}, t_{j-1}\rangle, \langle t_j, c(\overline{z})\rangle, \langle x_{j+1}, t_{j+1}\rangle, \ldots, \langle x_m, t_m\rangle\}$$

giving us $\Sigma[\phi'] : \Gamma'[\phi'], (\exists \overline{y}.\overline{U} \wedge \overline{R_P(\overline{q})^{@^i}})[\phi'] \longrightarrow F[\phi']$. Note that ϕ' is an mgu for the unification problem $\{\langle R_P(\overline{t}), R_P(x_1, \ldots, x_{j-1}, c(\overline{z}), x_{j+1}, \ldots, x_m)\rangle\}$.

Let θ be an mgu for $R_P(s_1, \ldots, s_m)$ and $R_P(\overline{t})$. This unifies $\{\langle s_1, t_1\rangle, \ldots, \langle s_m, t_m\rangle\}$. Let ρ be a substitution $\{\langle x_1, s_1\rangle, \ldots, \langle x_m, s_m\rangle\}$, that is, a substitution that will change the conclusion of the proxy rule for R into the conclusion of the particular rule the module N introduced for R in which we are interested (*i.e.*, turn $R(x_1, \ldots, x_m)$ into $R(s_1, \ldots, s_m)$). Composing ρ and θ gives us another unifier for $\{\langle R_P(\overline{t}), R_P(x_1, \ldots, x_{j-1}, c(\overline{z}), x_{j+1}, \ldots, x_m)\rangle\}$. Then we can write $\theta \circ \rho$ as $\omega \circ \phi'$ for some substitution ω.

The definition clause in $Lang(C)$ for the rule from N has the form

$$\forall \overline{v}.R_P(s_1, \ldots, s_m) \triangleq \exists \overline{y}, \overline{w}.\overline{U}[\rho] \wedge \overline{R_P(\overline{q})}[\rho] \wedge \overline{W} \wedge \overline{R_P(\overline{r})}$$

where the variables in $\overline{y}$ and $\overline{w}$ are distinct and $\overline{W}$ and $\overline{U}$ are premises not containing R or R_P. This incorporates the premises of the rule for R introduced by N ($\overline{W}$ and $\overline{R(\overline{r})}$ using variables in $\overline{v}$ and $\overline{w}$) and those of the proxy rule for R ($\overline{U}$ and $\overline{R(\overline{q})}$ using variables in $\overline{x}$ and $\overline{y}$) appropriately substituted with the substitution changing the conclusion of the rule. One possible premise sequent for this clause in the case analysis for $\mathcal{S}'$ is

$$\Sigma'[\rho][\theta] : \Gamma'[\theta], (\exists \overline{y}, \overline{w}.\overline{U}[\rho] \wedge (\overline{R_P(\overline{q})}[\rho])^{@^i} \wedge \overline{W} \wedge \overline{R_P(\overline{r})^{@^i}})[\theta] \longrightarrow F[\theta]$$

We can assume the variables in θ are disjoint from those in $\overline{y}$ and $\overline{w}$ and propagate the substitution inward over the clause body to get

$$\Sigma'[\rho][\theta] : \Gamma'[\theta], \exists \overline{y}, \overline{w}.\overline{U}[\rho][\theta] \wedge (\overline{R_P(\overline{q})}[\rho][\theta])^{@^i} \wedge \overline{W}[\theta] \wedge \overline{R_P(\overline{r})^{@^i}}[\theta] \longrightarrow F[\theta]$$

Note that the variables in $\overline{x}$ cannot occur in F or Γ'; if they did, we could rename away from them. Then this sequent is equivalent to

$$\Sigma'[\rho][\theta] : \Gamma'[\rho][\theta], \exists \overline{y}, \overline{w}.\overline{U}[\rho][\theta] \wedge (\overline{R_P(\overline{q})}[\rho][\theta])^{@^i} \wedge \overline{W}[\theta] \wedge \overline{R_P(\overline{r})^{@^i}}[\theta] \longrightarrow F[\rho][\theta]$$

Because $\theta \circ \rho$ is equivalent to $\omega \circ \phi'$, we can rewrite this using ϕ' and ω instead of ρ and θ when the two are used together:

$$\Sigma'[\phi'][\omega] : \Gamma'[\phi'][\omega], \exists \overline{y}, \overline{w}.\overline{U}[\phi'][\omega] \wedge (\overline{R_P(\overline{q})}[\phi'][\omega])^{@^i} \wedge \overline{W}[\theta] \wedge \overline{R_P(\overline{r})^{@^i}}[\theta] \longrightarrow F[\phi'][\omega]$$

Finally, via appropriate proof circumlocutions,[1] we find we have a proof of this sequent if we have one of the sequent

$$\Sigma'[\phi'][\omega] : \Gamma'[\phi'][\omega], (\exists \overline{y}, \overline{w}.\overline{U} \wedge (\overline{R_P(\overline{q})}^{@^i}))[\phi'][\omega] \longrightarrow F[\phi'][\omega] \tag{4.2}$$

But this is $\mathcal{S}_2$, for which we have a proof, with the substitution ω applied to it. Then Theorem 3.4 guarantees we have a proof of this sequent as well in the context of the language $Lang(C)$ and possibly using the lemmas in $\mathscr{L}$.

This argument applies to any mgu θ, so it applies to the one used to create the premise sequent $\mathcal{S}''$, finishing our proof. ■

Having shown we can build a proof for any case for an independent rule, we lay aside these ideas as somewhat completed for now, taking them up again in Section 4.4.4 where we describe how to build a composed proof of any property from the modular proofs written for it. We turn first to showing the $addP(R)$ property holds for a relation, a necessary component for the soundness of proofs using the proxy versions of their key relations.

Showing the Proxy Rule Subsumes Extension-Introduced Rules

As mentioned previously, a module introducing a proxy rule in $\mathbb{Q}$ is promising any actual rules introduced by modules building on it will be subsumed by it. This promise is fulfilled by introducing and proving the $addP(R)$ property, which each extension module adding to the definition of R must then prove holds for its new rules by writing a modular proof.

[1] The proof can end with the *cut* rule. The left branch of this proof rule uses a single formula as the context for the sequent, $\exists \overline{y}, \overline{w}.\overline{U}[\phi'][\omega] \wedge (\overline{R_P(\overline{q})}[\phi'][\omega])^{@^i} \wedge \overline{W}[\theta] \wedge \overline{R_P(\overline{r})^{@^i}}[\theta]$, the actual instantiated rule body formula, to derive the expected rule body formula $\exists \overline{y}, \overline{w}.\overline{U}[\phi'][\omega] \wedge (\overline{R_P(\overline{q})}[\phi'][\omega])^{@^i}$. This can be accomplished via appropriate uses of the $\exists\mathcal{L}$, $\exists\mathcal{R}$, $\wedge\mathcal{L}$, and $\wedge\mathcal{R}$ rules. The right branch of the *cut* rule is then Sequent 4.2.

We introduced the $addP(R)$ property above, specifying it has the form

$$\forall \overline{x}.R(\overline{x}) \supset F_R \supset R_P(\overline{x})$$

where F_R is a formula specific to the $addP(R)$ property for the particular relation R. This formula describes the conditions under which a derivation of the proxy version of a relation is expected to exist. These are generally well-formedness conditions for the arguments to R. For example, if the proxy rule uses a projection of R's primary component and that projection relation can depend on typing, F_R might include a condition that R's primary component be typable. By limiting the situations where R and R_P are expected to be equivalent, requiring equivalence only when F_R holds, we free extensions to write rules defining R more naturally, without building in the side conditions found in F_R that are required to show R_P will hold.

Any proof of the $addP(R)$ property will suffice for building full proofs of properties. We present a specific approach here to demonstrate how it *might* be done in common cases, as an appropriate proof can be subtle when the proxy rule contains a projection. Consider a common form of the proxy rule:

$$\frac{\overline{U} \qquad proj\,(\overline{t}, x_i, y) \qquad R(s_1, \ldots, s_{i-1}, y, s_{i+1}, \ldots, s_n)}{R(x_1, \ldots, x_n)}$$

This rule uses a projection ($proj$) with some arguments $\overline{t}$ and projects the primary component x_i of the conclusion to a fresh variable y and has a derivation of R for the projection y, in addition to some other premises $\overline{U}$ not using R. The most obvious approach to proving the $addP(R)$ property is to induct on $R(\overline{x})$; however, this does not generally give us a handle for demonstrating the proxy version of the relation will hold for the projection of the primary component.

While the language for our running example does not include a construct where induction on R alone would be a problem for proving $addP(R)$, there is another simple construct an extension might add that does demonstrate the issue. Consider a repeat-while loop

repeatWhile(b, c) repeating the body b of the loop while the condition c is true. This may project to a sequence of the body followed by a while loop for the remaining executions (*i.e.*, *seq*$(b,$ *while*$(c, b))$). If we want to show the proxy version of statement evaluation holds for this, we find we need to show the proxy version of evaluation for the projection of the repeat-while holds. This is not a sub-derivation of the evaluation of the repeat-while loop, so a proof by induction on R alone would not be helpful.

We propose utilizing another relation, built based on R in a way similar to how the proxy version of R is built, with this new relation building in an induction measure we can use for projections. We can then split the proof into two pieces, one building a derivation of the new relation from the derivation $R(\overline{x})$, and one building a derivation of $R_P(\overline{x})$ from the new relation. We call this new relation the *extension size version of the relation*, as it builds in a count of the number of places in the derivation where the rule used corresponds to one introduced by an extension module. Inducting on both the derivation of R and this count will generally give us a way to use the induction hypothesis for the projection as well.

Definition 4.22 (Extension size version of a relation)**.** *Let M be a module introducing a relation R and let C be either M itself or a module building on M. The extension size version of the relation R, written R_{ES}, is defined by making a modified version of each rule defining R. There are two classes of rules:*

- *For a rule in $\mathbb{R}^M$ or a default rule in $\mathbb{S}^M$ of the form*

$$\frac{\overline{R(\overline{s})} \qquad \overline{U}}{R(\overline{t})}$$

where $\overline{R(\overline{s})}$ denotes a set of premises built by R and $\overline{U}$ denotes a set of premises of other forms, we have a rule defining R_P as

$$\frac{\overline{R_P(\overline{s}, n_i)} \qquad \overline{U} \qquad n = \Sigma n_i}{R_{ES}(\overline{t}, n)}$$

in the same rule set. This rule replaces R with R_{ES} and sums the sizes of the premises.

- *Suppose we have a rule introduced by $\mathbb{R}^N$, where N builds on M, of the form*

$$\frac{\overline{R(\overline{s})} \qquad \overline{U}}{R(\overline{t})}$$

where $\overline{R(\overline{s})}$ denotes a set of premises built by R and $\overline{U}$ denotes a set of premises of other forms. We have another rule

$$\frac{\overline{R_{ES}(\overline{s}, n_i)} \qquad \overline{U} \qquad n = 1 + \Sigma n_i}{R_{ES}(\overline{t}, n)}$$

in $\mathbb{R}^N$. This rule replaces R with R_{ES} and sums the sizes of the premises, adding one as this is another extension-introduced rule.

These are the only rules in the definition of R_{ES}.

We can split the proof of $addP(R)$ into proofs of two separate formulas:

$$\forall \overline{x}.R(\overline{x}) \supset \exists n.R_{ES}(\overline{x}, n)$$
$$\forall \overline{x}, n.R_{ES}(\overline{x}, n) \supset F_R \supset R_P(\overline{x})$$

The first formula clearly holds, as we are basically adding summations to the existing rules for R to create R_{ES}. We can build an inductive proof of it mechanically. The second part is similar to the $addP(R)$ property itself, but the use of the extension size version of R instead of R itself allows us to induct on not only the structure of the derivation, but the extension size n of it as well.

We propose modularly proving this property by a nested induction on the extension size n and its derivation $R_{ES}(\overline{x}, n)$, with the latter as its key relation. This induction structure will allow the induction hypothesis to be applied to sub-derivations of $R_{ES}(\overline{x}, n)$ with an extension size no larger than n, as well as to derivations of R_{ES} with an extension size less than n, even those that are not sub-derivations of the original, and that might have an actual height larger than the original derivation. In writing modular proofs of this property,

specifically in those cases for rules for the proxy version of the relation incorporating a proxy rule with a projection, the latter use of the induction hypothesis allows constructing a derivation of the extension size version of the relation for a projection, then converting it to a derivation of the proxy version of the relation. Intuitively, we can see a derivation of R for a projection of the primary component should have a smaller extension size, as we have projected away the top-level constructor introduced by an extension, making this induction structure useful.

Consider again proving $addP(R)$ for a repeat-while loop. Because we are inducting on the extension size of the evaluation in addition to the derivation of evaluation itself, we can show the while loop to which it projects has an extension size that is strictly smaller than that of the repeat-while. The extension size is smaller because each use of the evaluation rules for the repeat-while loop that incremented the size, of which there must be at least one, has been replaced by uses of the rules for evaluating *seq* and *while* that do not increment the size. Then, because the size is smaller, we can use the induction hypothesis for the smaller size to show we have a derivation of the proxy version of evaluation for the projection. Thus induction on the extension size of the derivation of R along with induction on the derivation of R itself may allow us to prove $addP(R)$ in some situations where induction on the derivation alone does not.

Full Requirements for Modularly Proving Properties

We have seen certain situations will require us to use the proxy version of a relation in building a full proof, which means the proxy version of the relation must exist and it must hold when the preconditions of the property are met. This means, in certain situations, the key relation of a property must have a proxy rule in the set of proxy rules given by the module introducing the key relation. In those same situations, the set of lemmas available for use in the proof must also include the $addP(R)$ and $dropP(R)$ properties. Additionally, the $addP(R)$ property must be applicable to the assumptions of the property to be proven, as we will need to use it to lift the derivation of the key relation to its proxy version. We formalize these requirements with Definition 4.23.

Definition 4.23 (Valid lemma sets and proxy rule sets for properties)**.** *Let M be a module introducing a property P with key relation R. If M introduces R as well as P, any set of lemmas $\mathscr{L}$ and set of proxy rules $\mathbb{Q}^{M_R}$ is valid. If M does not introduce R, that is, if M imports the relation R from a module M_R (scenarios C and D in Figure 4.1), a set of lemmas $\mathscr{L}$ and a set of proxy rules $\mathbb{Q}^{M_R}$ are valid for proving P only if the following three conditions hold. First, there must be a rule defining R in $\mathbb{Q}^{M_R}$. Next, both $addP(R)$ and $dropP(R)$ must be in $\mathscr{L}$. Finally, the $addP(R)$ property must be applicable. That is, if P has the form $\forall\overline{x}.R(\overline{t}) \supset \forall\overline{x_1}.F_1 \supset \ldots \supset \forall\overline{x_n}.F_n \supset F$ then there is a proof of the sequent $\overline{x}, \overline{x_1}, \ldots, \overline{x_n} : R(\overline{t}), F_1, \ldots, F_n \longrightarrow R_P(\overline{t})$ assuming the variables in $\overline{x}, \overline{x_1}, \ldots, \overline{x_n}$ are unique to avoid inadvertent capture, with this proof possibly using the lemmas in $\mathscr{L}$.*

4.4.4 Completing the Proof Composition

Thus far, in this section we have described how we can use proofs of cases from the modular proofs to prove corresponding cases in the context of a composed language. Using our work thus far, we can describe our full proof composition. We split this into two separate cases, as the compositions have mildly different details. The first is that for properties where the key relation and its primary component category are introduced by the same module (scenarios A and B in Figure 4.1). These are the properties for which we use the key relation directly in the composed proof. The other is for properties where the key relation and its primary component category are introduced in separate modules (scenarios C and D in Figure 4.1). These are the properties for which we will use the proxy version of the key relation.

Theorem 4.24 (Basic proof composition soundness)**.** *Let M be a module introducing a property $P = \forall\overline{x}.R(\overline{t}) \supset F$ and its key relation R. Let C be a module building on M. Assume M has written a modular proof according to Definition 4.6 and each module $N \in \mathbb{B}^C \cup C$ that also builds on M has written a modular proof according to Definition 4.1, with each modular proof possibly using lemmas in $\mathscr{L}$. Then there is a proof of P in $Lang(C)$ possibly using lemmas in $\mathscr{L}$.*

Proof. The proof of P in $Lang(C)$ will have the canonical form of a proof for P. Note this

means we end the proof with the same uses of ind_m^i rules, as well as the same top-level case analysis on the key relation, as in writing the modular proofs. Each premise sequent from the top-level case analysis corresponds to a rule defining R in $Lang(C)$. Because the key relation R is introduced by the same module introducing the property, there are three possibilities for whence each rule came: it may have been introduced by M, it may have been introduced by a module building on M, or it may be an instantiation of the default rule from $\mathbb{S}^M$ for a constructor introduced by a module unrelated to M.

Lemma 4.10 tells us a rule introduced by M that unifies in the top-level case analysis in the context of $Lang(C)$ also unifies in the context of $Lang(\mathbb{M}(M, R))$, and that we have the same premise sequent corresponding to the rule in both contexts. Then we have a proof π of the same sequent as part of the modular proof written by module M. Lemma 4.9 lets us use this same proof π as our proof of the sequent in the context of $Lang(C)$. The same reasoning applies to sequents for rules introduced by modules building on M.

Lemma 4.14 ensures that if an instantiation of the default rule unifies with the hypothesis for the top-level case analysis in the composition, the default rule instantiated for ι unifies with it in the context of $Lang(\mathbb{M}(M, R))$, and with premise sequents $\mathcal{S}$ and $\mathcal{S}'$ such that $\mathcal{S} \sim^{\iota}_{c(\overline{y})} \mathcal{S}'$ holds for the constructor c for which the default rule is instantiated. Then we must have a proof of $\mathcal{S}$ as part of the modular proof written by M, and Lemma 4.13 tells us we can then build a proof of $\mathcal{S}'$ as well in the context of $Lang(C)$, as required for the full proof.

Since we can prove a premise sequent for each rule unifying with the key relation's derivation in the top-level case analysis, we have completed building our proof of the property for $Lang(C)$. ■

Theorem 4.25 (Proxy version proof composition soundness)**.** *Let M be a module introducing a property $P = \forall \overline{x}.R(\overline{t}) \supset F$ but not its key relation. Let C be a module building on M. Assume M has written a modular proof according to Definition 4.6 and each module $N \in \mathbb{B}^C \cup C$ that also builds on M has written a modular proof according to Definition 4.1, with each modular proof possibly using lemmas in $\mathscr{L}$. Further assume the requirements in*

Definition 4.23 are satisfied. Then there is a proof of P in Lang(C) using lemmas in $\mathscr{L}$.

Proof. The proof of P in *Lang*(C) will be split into two parts, a proof of the modified property $P' = \forall \overline{x}.R_P(\overline{t}) \supset F$ and a proof of the original property using the modified one.

The proof of P' will have the canonical form of a proof for P. We end the proof with the same uses of ind_m^i rules as the modular proofs, but the top-level case analysis is now on the proxy version of the key relation. Note the sequent that is the conclusion of the $def\mathcal{L}^{@^i}$ rule is related by $\sqsubseteq_R^i$ to the one for conclusion of the $def\mathcal{L}^{@^i}$ rule in a canonical form proof of P itself. The premise sequents from the top-level case analysis can be split into two groups.

The first group is for known rules, new rules, and instantiated default rules, that is, all the rules other than rules introduced by modules knowing the key relation but not the property. The conclusions of these rules have the form $R_P(\overline{s})$ where the original rules to which they correspond have the form $R(\overline{s})$. It is clear the former form unifies with $R_P(\overline{t})$ in exactly the same cases the latter unifies with $R(\overline{t})$, and with the same mgu. The rules for R_P are such that corresponding premise sequents are related by $\sqsubseteq_R^i$. Then Lemma 4.17 tells us there is a proof of the sequent for analysis on $R_P(\overline{t})$ if there is one for the sequent resulting from case analysis on $R(\overline{t})$. Depending on the provenance of the rule, whether or not it is an instantiated default rule, we can use either Lemma 4.10 or Lemma 4.14 to show a proof of the sequent exists in a modular proof, and either Lemma 4.9 or Lemma 4.13 to get a proof of the sequent for $R(\overline{t})$ in *Lang*(C). Therefore we can build a proof for the premise sequents for rules from these sources.

The second group is rules corresponding to rules defining R introduced by modules unrelated to M. The condition that there must be a rule defining the key relation R in the proxy rule set, part of Definition 4.23, guarantees a rule for κ was introduced as part of $\mathbb{K}(M, R)$ for the modular proof. Then Lemma 4.21, part 1, shows the modular proof written by module M must contain a proof π of a premise sequent for the rule introduced by $\mathbb{K}(M, R)$, since such a premise sequent will result from the top-level case analysis. Part 2 of Lemma 4.21 then gives us a proof of the sequent that appears in the proof in *Lang*(C).

Having proven P', we can use it to prove P. Specifically, we can use the proof that

$addP(R)$ is applicable to the premises of this property, guaranteed to exist by Definition 4.23, to build a derivation of $R_P(\overline{t})$. With this we can apply P' to complete the proof. ■

Having proven we can build a full proof of a property based on modular proofs in either case, we can combine these two facts into one as a formality.

Theorem 4.26 (Unified proof composition soundness)**.** *Let M be a module introducing a property $P = \forall \overline{x}.R(\overline{t}) \supset F$ and let C be a module building on M. Assume M has written a modular proof according to Definition 4.6 and each module $N \in \mathbb{B}^C \cup C$ that also builds on M has written a modular proof according to Definition 4.1, with each modular proof possibly using lemmas in $\mathscr{L}$. Further assume the requirements in Definition 4.23 are satisfied. Then there is a proof of P in Lang(C) possibly using lemmas in $\mathscr{L}$.*

Proof. If M also introduces the property's key relation R, we can apply Theorem 4.24 to build a proof of P. If R is introduced separately, we can apply Theorem 4.25 to build a proof of P. ■

4.5 Mutual Induction

Theorem 4.26 shows it is possible to create a composed proof of a property of the form $\forall \overline{x}.R(\overline{t}) \supset F$ for any language composition if each module in the composition gives an appropriate modular proof. However, proving properties individually is often not sufficient for the properties we want to prove for complex languages. In some languages we need to prove the properties we want by mutual induction, using the induction hypotheses from a set of properties to prove each property. The logic $\mathcal{G}$ supports mutual induction, so we can prove a set of n mutually-inductive properties by proving a formula of the form

$$(\forall \overline{x_1}.R_1(\overline{t_1}) \supset F_1) \wedge \ldots \wedge (\forall \overline{x_n}.R_n(\overline{t_n}) \supset F_n)$$

A proof of a formula like this using mutual induction permits us to use the induction hypothesis for any of the properties while proving each one of them.

However, proving mutually-inductive properties that are declared together in the same module is not always sufficient. Extension modules may find properties they want to prove need to be part of existing mutually-inductive groups as well. This might be the case if an extension introduces a new relation that is defined mutually recursively with existing relations. Then what we want is, if a module imported a mutually-inductive set of properties proven as $P_1 \wedge \ldots \wedge P_n$, for it to be able to provide a modular proof of $P_1 \wedge \ldots \wedge P_n \wedge P'_1 \wedge \ldots \wedge P'_m$ for new properties $P'_1, \ldots, P'_m$. Furthermore, we want a guarantee that a composed proof proving all the properties in this set, including also any introduced by other extension modules to be added to this set, exists for any composed language.

We consider both these scenarios in this section. We first consider lifting our concepts of modular proofs and composition from the preceding sections to the situation where a single module introduces a set of mutually-inductive properties. After that, we further lift our framework to the situation where extension modules may add new properties to preexisting sets of mutually-inductive properties. Both situations proceed much as expected. As seen in Theorems 4.24 and 4.25, what is important in constructing a composed proof from modular proofs is the proofs of individual cases from the top-level case analysis. The same is true in the mutually-inductive setting. At a high level, modular proofs for mutually-inductive properties are written in the same way as for individual properties, and under the restrictions of Definitions 4.1 and 4.6; the main difference is the presence of more than one induction hypothesis. The composition process then has proofs of the necessary proof cases to construct a full proof of the full set of properties.

There are some technical subtleties in extending our framework. These are primarily due to the possible inclusion of more generic constructs in the language used to write a modular proof for a property in a mutually-inductive set than if it were written alone, since the generic modules needed for all properties must be included in the language. The use of the proxy version of the key relation for lifting properties in the mutually-inductive setting is also slightly more complex than we saw in the previous section. All properties in a mutually-inductive set must use the proxy version of their key relations if one does, whether they would need to use it as an individual property or not. We sketch the changes

$$\frac{\begin{array}{l}\Sigma : \Gamma, \quad \forall \overline{x_1}^1.F_1^1 \supset \cdots \supset \forall \overline{x_m}^1.(A^1)^{*^i} \supset F^1, \ldots, \\ \qquad \forall \overline{x_1}^n.F_1^n \supset \cdots \supset \forall \overline{x_m}^n.(A^n)^{*^i} \supset F^n \longrightarrow \\ \qquad\qquad (\forall \overline{x_1}^1.F_1^1 \supset \cdots \supset \forall \overline{x_m}^1.(A^1)^{@^i} \supset F^1) \wedge \ldots \wedge \\ \qquad\qquad (\forall \overline{x_1}^n.F_1^n \supset \cdots \supset \forall \overline{x_m}^n.(A^n)^{@^i} \supset F^n)\end{array}}{\begin{array}{l}\Sigma : \Gamma \longrightarrow \quad (\forall \overline{x_1}^1.F_1^1 \supset \cdots \supset \forall \overline{x_m}^1.A^1 \supset F^1) \wedge \ldots \wedge \\ \qquad\qquad (\forall \overline{x_1}^n.F_1^n \supset \cdots \supset \forall \overline{x_m}^n.A^n \supset F^n)\end{array}} \; \mathit{mutind}_m^i, \text{ each } A^j \text{ is atomic}$$

Annotations of the form $*^i$ and $@^i$ must not already appear in the conclusion sequent

Figure 4.3: Generalized induction rule for mutual induction

to the framework and proofs of its correctness in the remainder of this section.

4.5.1 Modularly Proving Mutually-Inductive Property Sets

To prove mutually-inductive property sets, we must first understand mutual induction in $\mathcal{G}$. The logic $\mathcal{G}$ has a generalized version of the ind_m^i rule from Figure 3.3, mutind_m^i, shown in Figure 4.3.[2] As in the general ind_m^i rule, the mutind_m^i rule uses $@^i$ and $*^i$ annotations to mark derivations of the original size and smaller sizes respectively. The difference is we now have n induction hypotheses generated, one for each mutual theorem, rather than a single one. Reasoning about sequents in proofs ending with the mutind_m^i rule is the same as reasoning about any other sequent. Note that all our earlier results, including Theorem 3.4, are still valid with mutual induction included in the logic.

For individual properties, in Section 3.4, we saw the canonical form of a proof ends at the root with some uses of the ind_m^i rule to induct on the key relation and other premises, uses of the $\forall\mathcal{R}$ and $\supset\mathcal{R}$ rules to introduce eigenvariables and turn premises into hypotheses, and a use of the $\mathit{def}\mathcal{L}^{@^i}$ rule for the top-level case analysis. We need to modify this slightly for situations using mutual induction. Instead of using the ind_m^i rule, the canonical form of a proof of mutually-inductive properties uses the mutind_m^i rule, once again using it to induct on as many premises as it wants. The premise sequent of the uses of the induction rule will have a conclusion of the form $P_1 \wedge \ldots \wedge P_n$. We use the $\wedge\mathcal{R}$ rule to split this into one

[2] For simplicity of presentation, we assume induction on the m^{th} premise in each property. This may be generalized to different premise indices for different properties in the set.

sequent for proving each individual property. The proofs of these property-specific sequents then use the $\forall\mathcal{R}$ and $\supset\mathcal{R}$ rules, followed by the $def\mathcal{L}^{@^i}$ rule for the property's top-level case analysis.

We can distribute the cases from the top-level case analysis for each property in the set as if we were proving each property individually. For modular proofs written by modules building on the one introducing the set of properties, the modular proof in a module M can be written for $Lang(M)$, as specified in Definition 4.1. The situation for modules introducing sets of properties is a bit more complicated. In a set of properties, we can have properties with different key relations, and thus properties from different scenarios illustrated in Figure 4.1. For example, we could introduce a property with a key relation and primary component category introduced by the current module, scenario A in Figure 4.1, along with a property fitting scenario B, where the primary component category is introduced by an imported module. Thus there is no relation R such that $Lang(\mathbb{M}(M, R))$ is valid for proving a set of properties introduced by module M, as different properties might require different generic constructors. However, we can appropriately satisfy the requirements of Definition 4.6 if we prove each case from each property's top-level case analysis, obeying the appropriate restrictions on the proofs, with a language that *includes* the prescribed language for it, having all the constructors and rules expected. Thus we can write our modular proof for a set of mutually-inductive properties with a set of key relations S for the language

$$\bigcup_{R \in S} Lang(\mathbb{M}(M, R))$$

where the union of languages is the union of the individual sets in the 4-tuples defining them. This language includes the required generic constructors for the modular proof cases for each property. The restrictions on proofs imposed by Definition 4.6 ensure the extra generic constructors and rules for them cannot affect the proofs of cases needed for a composed proof, so we are able to use the proof of each case in constructing the composed proof for the set as we can for individual properties.

While the extra generic constructors and rules cannot affect the proofs of necessary

cases, they may introduce cases from the top-level case analysis on R that would not be present in the context of $\mathit{Lang}(\mathbb{M}(M, R))$ alone. For example, suppose we have modules M and N where M builds on N. Suppose N introduces a relation R and M introduces R', both having the same primary component category. Module M then introduces mutually-inductive properties P with key relation R and P' with key relation R'. Property P has the key relation introduced in the same module as its primary component category, fitting scenario C in Figure 4.1, and P' has them introduced in different modules, fitting scenario B. Because P fits scenario C, the language $\mathit{Lang}(\mathbb{M}(M, R))$ has a generic constructor κ. The default rule for relation R' is instantiated for this generic constructor, and thus we have an extra case from the top-level case analysis for proving P', one which is not needed for any composition. In this situation, it is equivalent to the case for the generic case for instantiating the default rule of R' for ι, which is needed for the composition, and thus can be given the same proof. However, because such a case is unnecessary, in practice we may skip proving it, giving us a modular *partial* proof that may have unneeded cases skipped.

Recall from Section 4.4.3 that we need to use the proxy version of a relation for constructing the compositions of some properties, those where a property is introduced in a different module than its key relation (scenarios C and D in Figure 4.1). In order to use the proxy version of a relation in a mutually-inductive setting, we need to modify its definition slightly. In Definition 4.15, we identified the proxy version of a relation R, written R_P, as having rules corresponding to those for R, but with premise derivations of R changed to be premise derivations of R_P. Additionally, rules introduced by extensions to the module introducing R have the premises of the proxy rule for R added to them, these premises also being modified to use R_P. For mutual induction, we define the proxy version of a *set* of relations. We add the premises of the proxy rule as in Definition 4.15, but now we modify premises deriving any relation in the set to use their proxy version. We relax the restriction from Definition 4.15 that a relation R for which we define the proxy version is not defined mutually-recursively with any other relation for sets. Each relation in a set must be defined mutually-recursively only with others in the set; the restriction for a single relation R is exactly the case where the set is $\{R\}$. We similarly expand the definition of the extension

size version of a relation, R_{ES}, to sets of relations.

The purpose of the proxy version of a relation is to make the proof of the generic case using the proxy rule for a relation in the modular proof correspond to the actual cases that occur in a composed proof. In our proof of composition soundness for individual properties using the proxy version of the key relation, we defined the proxy version of a sequent (Definition 4.16) to have each annotated derivation $R(\bar{t})^{*^i}$ or $R(\bar{t})^{@^i}$ replaced with its proxy version, $R_P(\bar{t})^{*^i}$ or $R_P(\bar{t})^{@^i}$. This ensures the induction hypothesis in the composed language, using R_P, can be used with the hypotheses corresponding to those with which it could be used in the modular proof.

In the mutually-inductive setting, where we can have multiple induction hypotheses, we need to ensure the same thing for all of them. If some properties used the proxy version of their key relations and some didn't, in the composition, we could end up with an induction hypothesis for $R_P(\bar{t})^{*^i}$ but a hypothesis to which we want to apply it of the form $R(\bar{t})^{*^i}$. Then some uses of the induction hypothesis from the modular proof might fail, and the composed proof would also fail. Thus, in creating the composed proof of the properties, we have all properties use the proxy versions of their key relations if one does. Additionally, these proxy versions must all be defined together, using the same set of relations to determine which premises must be modified to their proxy versions.

We can now sketch a proof of the existence of a composed proof for a set of mutually-inductive properties.

Theorem 4.27 (Mutual proof composition soundness)**.** *Let M be a module introducing a set of mutually-inductive properties $\{P_1, \ldots, P_n\}$, with key relations in the set S. Let C be a module building on M. Assume M has written a modular proof according to the restrictions in Definition 4.6 and using language $\bigcup_{R \in S} Lang(\mathbb{M}(M, R))$, with the proof possibly using lemmas in $\mathscr{L}$. Assume also each module $N \in \mathbb{B}^C \cup C$ that builds on M has written a modular proof for the set according to Definition 4.1, also possibly using lemmas in $\mathscr{L}$. Further assume the requirements in Definition 4.23 are satisfied for each property in the set if any one uses an imported key relation. Then there is a proof of $P_1 \wedge \ldots \wedge P_n$ in $Lang(C)$ possibly using lemmas in $\mathscr{L}$.*

Proof. As in Theorem 4.26, we can split the proof depending on whether we need to use the proxy version of the key relations, that is, whether any one of the properties in the set has an imported key relation.

If all the key relations are introduced by M, we do not need the use the proxy version of the key relations. In this case, the proof is similar to Theorem 4.24. The composed proof will have the canonical form used for the modular proofs, so each property needs a proof for the cases arising from its top-level case analysis. Because the modular proofs contain the same proof cases as if we were proving properties individually, there is a corresponding modular case, either known or generic, for each case in the composed proof. As noted above, the restrictions on the modular proof in the introducing module ensure the addition of other generic constructors in the language used for writing the modular proof cannot affect property proofs, so we can lift the cases to the composed language as in Theorem 4.24. Then each case in the composed proof has a corresponding sub-proof of a modular proof, and we can complete the construction.

If one of the properties is imported, the proof is similar to Theorem 4.25. As there, we split the proof into two parts, proving the properties using the proxy versions of their key relations, then using the modified properties to prove the ones we want. This proof has the canonical form for the property set, but with the top-level case analysis for each property being on the proxy version of its key relation. As in the previous case, each sequent in the composed proof has a corresponding one in some modular proof, and we can use the same arguments as in Theorem 4.25 to lift them to the composed language. Then we have a proof for each case in the composed proof, completing it. ■

We can see that Theorem 4.26 is a corollary of Theorem 4.27 where the set of properties being proven contains a single property.

4.5.2 Extending Mutually-Inductive Sets of Properties

As in the previous situations, both for individual properties and for mutually-inductive sets all introduced by the same module, what matters most for crafting a composed proof when

properties may be added to existing mutually-inductive sets is that each proof case arising from the top-level case analysis in the composition has a corresponding proof case proven in some modular proof. To see how this works, we need to consider the canonical form of a proof of a set of properties, the languages in which modular proofs are written, and the proxy versions of relations and their use in proofs.

If an extension module adds new properties, the existing canonical form for the proof will not work. In proving $P_1 \wedge \ldots \wedge P_n$, the canonical-form proof ends with some uses of the mutind^i_m rule, uses of the $\wedge\mathcal{R}$ rule to split the properties, and property-specific uses of the $\forall\mathcal{R}$, $\supset\mathcal{R}$, and $\mathit{def}\mathcal{L}^{@^i}$ rules. However, in proving $P_1 \wedge \ldots \wedge P_n \wedge P'_1 \wedge \ldots \wedge P'_m$, the exact same structure doesn't work because the splitting of the conjunctions with the existing $\wedge\mathcal{R}$ rules doesn't split them all. To modify the set-up for the canonical-form proof to accommodate added properties, we add extra applications of the $\wedge\mathcal{R}$ rule sufficient to split all the properties, as well as property-specific uses of the $\forall\mathcal{R}$, $\supset\mathcal{R}$, and $\mathit{def}\mathcal{L}^{@^i}$ for the new properties. Using this modified proof form, the sequent for the proof of each property has the same form as under the unmodified one, but with more induction hypotheses.

When writing a modular proof for a set of properties imported into a module M, we write it in the context of $\mathit{Lang}(M)$. However, when proving a set of newly-introduced properties with key relations in S in a module M, we write the modular proof in the context of $\bigcup_{R \in S} \mathit{Lang}(\mathbb{M}(M, R))$. When we add new properties to an existing set, we write a modular proof in the language

$$\mathit{Lang}(M) \cup \bigcup_{R \in S} \mathit{Lang}(\mathbb{M}(M, R))$$

where the new properties have key relations in S. Note that $\mathit{Lang}(M)$ is included in $\mathit{Lang}(\mathbb{M}(M, R))$ for any relation R; by including it separately, we generalize this to subsume the case where no new properties are added. This new language includes all the constructs and rules from the language expected for individual modular proofs both for the imported properties and for the new ones.

The modular proof for a set of properties with new additions follows the restrictions

in Definition 4.6, which are a superset of those in Definition 4.1. As noted previously, the restrictions in these definitions prevent using rules from generic extensions not part of the set needed for proving an individual property in the set. As in the case for mutually-inductive properties without extension additions, using a language with extra generic constructs can create unneeded cases from the top-level case analysis, but skipping these and creating modular partial proofs is still sound for creating the composition.

When one property in a mutually-inductive set uses an imported key relation, meaning it needs to use the proxy version for constructing the composed proof, all properties in the set need to use the proxy versions of their key relations in the composition. This then imposes a requirement for all properties to satisfy the requirements of Definition 4.23. This is also true when we can add new properties to a set: all properties, both new and imported, need to satisfy the requirements of Definition 4.23. Because of this, we only allow adding a property that uses an imported key relation to an existing property set if a property already in the set was introduced in a different module than its key relation (*i.e.*, an existing property used an imported key relation when it was introduced). In that case, we know all properties in the set in a composed language will satisfy Definition 4.23, since each extension module that may add properties will know it must satisfy it. If some extension imposed the necessity of Definition 4.23 for a set of properties by introducing a property with an imported key relation in an extension, other extensions would not prove it for their properties, and the composition would fail.

Having discussed the changes necessary for handling the addition of properties to existing sets, we can now prove a composed proof of the full set from all included modules exists for a composed language.

Theorem 4.28 (Added properties composition soundness)**.** *Let M be a module introducing a set of mutually-inductive properties $\overline{P}$ with key relations in S and let C be a module building on M. Assume M has written a modular proof according to the modified version of Definition 4.6 using $\bigcup_{R \in S} Lang(\mathbb{M}(M, R))$. Assume also that each module $N \in \mathbb{B}^C \cup \{C\}$ building on M adds a (possibly empty) set of properties $\overline{P_N}$ to the set $\overline{P}$, the new properties having key relations in S_N, and writes a modular proof for the combined set using*

$Lang(M) \cup \bigcup_{R \in S_N} Lang(\mathbb{M}(M, R))$ as the language obeying the restrictions of Definition 4.6, using a canonical form for the proof that extends module M's canonical form as specified above. Assume all modular proofs are written possibly using lemmas in $\mathscr{L}$. Assume also the requirements of Definition 4.23 are satisfied for each property. Then there is a proof of the full set of mutually-inductive properties $\overline{P} \cup (\bigcup_{N \in \mathbb{B}^C \cup \{C\}} \overline{P_N})$ in $Lang(C)$ possibly using lemmas in $\mathscr{L}$.

Proof. The proof is similar to that of Theorem 4.27. As noted above, the modifications to the canonical form for the proof leave the sequents for each property the same, other than adding new induction hypotheses. Also, the addition of extra generic constructs to the languages used for writing modular proofs don't affect the proof cases needed for constructing a composed proof. Then the reasoning from Theorem 4.27 applies here as well. ■

As Theorem 4.26 is a corollary of Theorem 4.27, so Theorem 4.27 is a corollary of Theorem 4.28 in the case where no module adds to the set of properties.

This concludes our discussion of proving individual properties and mutually-inductive sets of properties in a modular fashion. We have shown how modules can declare properties, specifying the canonical forms their proofs must take. We have also shown that, when each module included in a composition that knows a property, or set of properties, writes a modular proof of this form, we can construct a composed proof for the composed language, demonstrating the property, or set of properties, hold for the composed language. Thus we see that modular proofs can be used to guarantee metatheoretic properties hold for any composed language.

Chapter 5

Composing the Metatheory of Modules

In the previous chapter, we discussed proving individual properties and sets of mutually-inductive properties using sets of lemmas assumed to be true. However, we are interested in proving full sets of metatheoretic properties, not individual properties or mutually-inductive groups of them alone. In proving a set of properties soundly, property P_1 may be used freely as a lemma in proving property P_2 only if P_1 can be proven without knowing P_2 holds. If the proofs of P_1 and P_2 both depend on the other holding, we have implicitly assumed each is true, without evidence, in proving it is true.

Formally, in proving a set of properties in a sound manner, we need an order on the properties that is asymmetric and irreflexive, where, when proving a property P, we may use as a lemma any property P' where P' comes before P in the order. Recall from Chapter 3 that lemma use is justified by the *cut* rule, and that using a lemma with a formula L requires a derivation of $\varnothing : \varnothing \longrightarrow L$. Only using a property as a lemma when an asymmetric and irreflexive order specifies it is less than the current one ensures such a derivation exists.

The difficulty in the extensible setting is in ensuring we can compose all the modular proof orders to form a single asymmetric, irreflexive order for any composed language, and that the composed order reflects the modular ones used in writing all the modular proofs from all included modules. This is necessary for creating full sets of proofs for full sets of properties for a composed language. Consider the module structure shown in Figure 5.1. We have a host language module H with modules A and B building on it independently of one another. We then have two modules C and D both building on A and B, but independent of one another, and a final module E building on all of them. As in prior diagrams, we

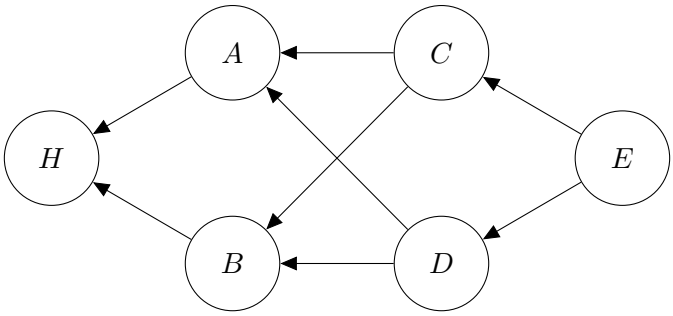


Figure 5.1: Module structure example for order composition

elide the transitive builds-on arrows. Suppose module A introduces a property P_A and module B introduces a property P_B. Suppose also we want to compose the metatheory of $Lang(E)$ to create full proofs of all properties from all modules in the figure. Whether this is possible depends on how C and D treated P_A and P_B in ordering. If C used P_A as a lemma in proving P_B, and D used P_B as a lemma in proving P_A, we cannot compose the proofs. Both C's modular proofs and D's modular proofs can be valid in the context of their individual modules, but their orderings of the imported properties are incompatible when put together, as both P_A must be proven before P_B and P_B must be proven before P_A.

The key to a completely modular solution to this problem will be having aspects of the ordering dictated by choices made by H, A, and B, the modules on which C and D build. These choices can specify whether P_A can be used as a lemma in proving P_B, P_B can be used as a lemma in proving P_A, or neither. Having the relative ordering of the properties decided by the modules on which C and D, the modules bringing the properties together, build ensures they must agree on an ordering. If both modules C and D can be written respecting this ordering, then module E can combine them. If not, at least one of them is ruled out because it cannot prove what it wants to prove, if not both being ruled out for that reason. In this case, the impossibility of writing C or D as desired is known at the time of attempting to write the module, so it cannot be contributed to the language library when it cannot compose with other modules. Thus all modules that are part of a library can be composed, even if ones we might wish to write cannot be part of the library.

In the rest of this chapter, we address the problem of composing property orders and ensuring an asymmetric and irreflexive order exists for any composition of modules, and how we can use this to ensure all properties hold for a composed language. We first explain the requirements for sound, completely modular property order composition schemes in Section 5.1. We also prove we can build full sets of sound property proofs for all properties of a composed language if the property order composition follows the requirements we lay out. We then look at two different schemes fulfilling these requirements and discuss their benefits and drawbacks in Sections 5.2 and 5.3. These schemes permit H, A, and B to dictate the relative ordering of P_A and P_B, even though the modules A and B that introduce them are unrelated, thus ensuring C and D cannot disagree on the ordering. Finally, Section 5.4 looks at a scheme that does not guarantee any arbitrary set of modules is composable, but which may still be desirable in some situations, and discuss this as an alternative to the other choices that are completely modular.

5.1 Requirements for Sound Metatheory Composition

In our reasoning framework, we have a *metatheory description tuple* $\langle \mathscr{P}, <, \mathbb{Q} \rangle$ for a module, where $\mathscr{P}$ contains the properties known to the module, $<$ is an order for those properties, and $\mathbb{Q}$ is the set of proxy rules as described in Section 4.3.2. Formally, $\mathscr{P}$ is a set of sets of one or more mutually-inductive properties. We will require a well-formed property set to have disjoint sets within it, that is, $\mathscr{P}$ cannot contain sets S and S' such that $S \cap S'$ is not the empty set. Furthermore, unlike what we saw with elements of the language specification tuple, $\mathscr{P}^M$ contains not only properties M introduces, but also those M imports from the modules on which it builds.

Sound metatheory composition requires that the set of sets of mutually-inductive properties for a module be compatible with those it imports. We define the notion of a property composition, combining the property sets from the modules on which one builds, that creates a new set of property sets fulfilling these desiderata. Property composition combines corresponding mutually-inductive sets of properties from different modules in such a way

that the composition contains all the properties from all the modules being composed. It also ensures properties that were in the same mutually-inductive set in one of the modules being composed are also in the same mutually-inductive set in the composition. We will require a module's property set to include the composed one, meaning it has a superset of each mutually-inductive set from the composition.

Definition 5.1 (Property set composition)**.** *The composition of the sets of properties from modules on which a module M builds, written $propertyCompose(M)$, is a set of sets of modules produced by the following algorithm. Start with $C := \varnothing$. For each set $S \in \mathscr{P}^N$, for each module $N \in \mathbb{B}^M$, consider whether S intersects with any set in C.*

- *If there is a set $S' \in C$ such that $S \cap S' \neq \varnothing$, update $C := (C \setminus S') \cup \{S' \cup S\}$.*
- *If not, update $C := C \cup \{S\}$.*

Then $propertyCompose(M)$ is the final set C after going through all mutually-inductive property sets from all the modules on which M builds.

Note that if the sets in each property set being composed (*i.e.*, all the sets in each $\mathscr{P}^N$ for $N \in \mathbb{B}^M$) are disjoint, then $propertyCompose(M)$ identifies a unique set of sets of properties, and all its element sets are disjoint as well.

The property order $<^M$ relates the sets of sets of properties in $\mathscr{P}^M$, giving us an order in which we may prove the sets of properties, using properties from earlier sets as lemmas in proving later ones. We require a well-formed order to be asymmetric and irreflexive, which makes using it for writing proofs sound. A module's property order is the one used for determining which properties may be used as lemmas in writing its modular proofs. Before discussing more about the specifics of property orders and their composition, we define a convenient notion we will use in further definitions. While an order $<$ relates sets of properties, we often want to refer to orders between particular properties in sets. We can extract an ordering directly for properties from one for sets of properties.

Definition 5.2 (Extracted direct property order)**.** *Let $<$ be an order on sets of properties. Its extracted direct property order is written $<_P$. We have $P <_P P'$ if and only if there are*

sets S and S' such that $P \in S$, $P' \in S'$, and $S < S'$ holds.

The composition of a set of property orders does not have only one choice for how to create it. Rather, we have requirements for what forms a good composition, which many schemes may satisfy. Each language library may choose one that all modules in the library use. Intuitively, what we want of a composition of property orders is for it to be asymmetric and irreflexive so it is sound for proving properties, and for it to maintain the relative orderings of properties from modules that are included in the composition so we have the same lemmas available when moving from modular proofs to composed ones. By the latter desideratum we mean that if we had $P <_P^N P'$ and $<$ is an order composed from several including $<^N$, then we should also have $P <_P P'$. In this section, we describe what is required of a scheme for composing property orders and what is required of the metatheory of modules relative to metatheory compositions of the modules on which they build. Based on these requirements, we prove modular proofs guarantee all properties, from all modules included in a composed language, hold for the composed language.

Similar to how we composed the property sets of the modules on which another builds, we can compose the property orders of the modules on which another builds. We will write $\mathit{orderCompose}(M)$ to signify the composition of the property orders $<^N$ for each module $N \in \mathbb{B}^M$. This order will relate the sets of properties in $\mathit{propertyCompose}(M)$. We define here what is required for a module's property order to be well-formed relative to an order composition scheme $\mathit{orderCompose}()$. The important point in this definition is that, in creating a module's property order, the module's author must maintain the relationships from the composition of the orders from the modules on which it builds. Essentially this requires using the composed order as a starting point, and possibly adding new relationships between sets of properties to it, but without adding any to make it symmetric or reflexive.

Definition 5.3 (Property order well-formedness)**.** *Let M be a module and $\mathit{orderCompose}()$ be the order composition scheme its language library uses. For M's property order $<^M$ to be well-formed, the order must be asymmetric and irreflexive. It must also include all the relationships of the composed order of the modules on which M builds; that is, if $<$ is the*

composed order orderCompose(M) *of the modules on which* M *builds and* $P <_P P'$*, it must be that* $P <_P^M P'$*.*

Note that this defines a minimum requirement for order well-formedness; particular composition schemes may add more requirements, as we shall see in Section 5.2.

The definition of order well-formedness does not yet address the issue of compositionality of module orders. Both modules C and D in our ordering example, shown in Figure 5.1, can have well-formed property orders by this definition, even if they order P_A and P_B differently. As long as their orders are asymmetric and irreflexive, and they respect the composed order of $<^H$, $<^A$, and $<^B$, they are well-formed. They could still order P_A and P_B differently if the order of these two properties is not somehow specified by the composition scheme used in the language library, the one relative to which C's and D's orders are well-formed. The solution to ensuring composability of any property orders for a language library, which will allow us to prove all properties from any module hold for a composed language, lies in our requirements on ordering schemes. These requirements specify compositions must be asymmetric and irreflexive, as well as maintain the orders included in them, as discussed as a desideratum of composition above. Finally, order composition must be total when the orders being composed are well-formed. This is the criterion that ensures the orders specified by C and D must compose.

Definition 5.4 (Completely modular order composition scheme)**.** *An order composition scheme orderCompose*() *is completely modular if it satisfies three criteria. In these, assume* M *is a module where each module* N *on which* M *builds (*$N \in \mathbb{B}^M$*) has a well-formed property order* $<^N$ *relative to orderCompose*()*, the ordering scheme used by the language library of which the modules are members.*

1. *The order composition orderCompose*(M) *is an asymmetric and irreflexive order.*
2. *For a module* N *included in the composition (*$N \in \mathbb{B}^M$*), if we have* $P <_P^N P'$ *and orderCompose*(M) *is* $<$*, we also have* $P <_P P'$*.*
3. *The order composition orderCompose*(M) *exists.*

The first and final criteria ensure any module can have a well-formed property order by Definition 5.3. The requirements for a module's order to be well-formed are that it be asymmetric and irreflexive, and that it maintains the composed order of orders from the modules on which it builds. The final criterion ensures the required composed order exists, and the first one ensures the composed order itself is asymmetric and irreflexive. Then all a module's author needs to do to have a well-formed property order is not add any new relationships between sets of properties that make the order symmetric or reflexive. Note this does not mean any module can have the property order its author *wants*, necessarily, just that a valid one exists. A completely modular ordering scheme will prevent at least one of modules C and D from having the order of properties it wants, and they may not be able to write their required modular proofs using it, but the composed order itself is well-formed for both C and D.

The second criterion for a property order composition scheme to be completely modular, that it maintains the relative ordering of properties from orders included in the composition, ensures the proof composition for $Lang(M)$ is sound for all the properties when using the well-formed order $<^M$ to order the use of lemmas, if each module used its own property order in writing modular proofs. Because each included order is maintained by the composition, and the composition is maintained by M's order, the properties available as lemmas in proving a property in each modular proof are then also available as lemmas using M's order. We can formalize this intuition in a theorem, but first we define well-formedness of a module's metatheory description tuple to utilize in stating the theorem.

Definition 5.5 (Metatheory description tuple well-formedness)**.** *Let M be a language module with metatheory description tuple $\langle \mathscr{P}^M, <^M, \mathbb{Q}^M \rangle$. This is well-formed if the following criteria hold:*

- *For each set S of properties in $propertyCompose(M)$, there is a set $S' \in \mathscr{P}^M$ such that $S \subseteq S'$. Furthermore, each set of properties in $propertyCompose(M)$ is disjoint from all the other sets.*

- *For each property P introduced by M (*i.e.*, P is in a set in $\mathscr{P}^M$ but not in any set in*

propertyCompose(M)*), the requirements of Definition 4.23 are satisfied for* P *with the set of proxy rules introduced by the module introducing its key relation and the set of lemmas being properties in sets* S' *in* $\mathscr{P}^M$ *where* $S' <^M S$ *(*i.e.*, the set of lemmas is* $\{P | P \in S' \wedge S' \in \mathscr{P}^M \wedge S' <^M S\}$*).*

- *The property order* $<^M$ *is well-formed by Definition 5.3 relative to the order composition scheme used by the language library.*

- *For each set of properties* S *in* $\mathscr{P}^M$*, there exists a valid modular proof according to Definition 4.1 or Definition 4.6 as appropriate, with modifications to extend these to full sets of proofs as given in Section 4.5. Furthermore, this modular proof uses as its set of lemmas properties in sets* S' *in* $\mathscr{P}^M$ *where* $S' <^M S$ *(*i.e.*, its set of lemmas is* $\{P | P \in S' \wedge S' \in \mathscr{P}^M \wedge S' <^M S\}$*).*

- *The set of proxy rules* $\mathbb{Q}^M$ *is well-formed by Definition 4.3.*

Using this definition of well-formedness for metatheory description tuples, we can fulfill our intuition from above, showing that all properties from all modules included in a composition hold for modules with well-formed metatheory descriptions that use a completely modular ordering scheme.

Theorem 5.6 (Metatheory composition soundness)**.** *Let* M *be a module with builds-on set* $\mathbb{B}^M$*. Let each module* $N \in \mathbb{B}^M \cup \{M\}$ *have a well-formed metatheory description tuple* $\langle \mathscr{P}^N, <^N, \mathbb{Q}^N \rangle$*, and let orderCompose*() *be a completely modular ordering scheme used by the language library. Then all properties introduced by the modules* $\mathbb{B}^M \cup \{M\}$ *have proofs for Lang*(M)*.*

Proof. Because $<^M$ is well-formed, it is asymmetric and irreflexive, and thus sound for proving properties. The first requirement for the well-formedness of M's metatheory description tuple ensures $\mathscr{P}^M$ contains each property from any module included in the composition. Then what needs to be shown is that we can prove each property in $\mathscr{P}^M$ using $<^M$ as the lemma order.

Consider a property set S in $\mathscr{P}^M$. Theorems 4.26, 4.27, and 4.28 show we can construct a full proof for the properties in S from the modular proofs written by the modules knowing it. These theorems rely on the modular proofs using the same lemma set as is used in the composition. Because the logic is monotonic, what is actually required is that the lemma set in the composition must be a superset of the one used for each modular proof. What we need to show to use these theorems then is that the necessary modular proofs exist, Definition 4.23 is satisfied when needed, and $<^M$ maintains the lemma sets used by each modular proof.

The first two parts come directly from the definition of well-formedness of the metatheory description tuples. Each module N must have the appropriate modular proofs for the sets of properties in $\mathscr{P}^N$. Each property in $\mathscr{P}^M$ also must be introduced by M or some module on which it builds, and that module must ensure the requirements of Definition 4.23 are satisfied for it with the set of lemmas used for its modular proof. Note Definition 4.23 requires certain properties be present in the set of lemmas, so it is also valid if we add more lemmas to the set.

All that remains then is showing the sets of lemmas used by each modular proof are maintained. That is, we need to show that for each module $N \in \mathbb{B}^M$, if a property P' known to N is a member of a set $S' \in \mathscr{P}^N$ and $S' <^N S$, then there is a set $S'' \in \mathscr{P}^M$ where $P' \in S''$ and $S'' <^M S$. This comes from the requirement that $<^M$ maintain the relationships between properties from $orderCompose(M)$ as part of its well-formedness requirements, and the second requirement for a completely modular ordering scheme, that $<^N$ be similarly maintained in $orderCompose(M)$. Then the lemma set an arbitrary module used for proving the properties in S is a subset of the one used in the composition. We can then apply the appropriate theorem from Chapter 4 to build a proof of the mutually-inductive set of properties S for $Lang(M)$ using $<^M$ as the property order.

This reasoning applies to all sets of properties in $\mathscr{P}^M$, so all properties from any module included in the composition have proofs for $Lang(M)$. ■

We turn now to look at two completely modular ordering schemes for properties, one

using partially-ordered sets of properties and one where we order properties via ordered tags. We close this chapter by looking at what happens if we do not require totality in an ordering scheme, the third requirement for completely modular ordering schemes, and why this might sometimes be desirable.

5.2 Partially-Ordered Property Sets

One approach to order composition is to have each module's order be a strict partial order over its known property sets. The composition, written $orderCompose_{PO}(M)$, is essentially the transitive closure of the union of the orders being composed. The formal definition is slightly more complicated, as the composed order relates elements in the composed property set, which may have changed from those in individual modules by other modules adding new properties to them.

Definition 5.7 (Partially-ordered property order composition)**.** *Let $<$ be a property order such that we have $S < S'$ for sets S and S' in $propertyCompose(M)$ if and only if there are sets S_0 and S'_0 in $\mathscr{P}^N$ for some module N on which M builds ($N \in \mathbb{B}^M$) and $S_0 <^N S'_0$. Then $orderCompose_{PO}(M)$ is the transitive closure of $<$.*

The order composition scheme $orderCompose_{PO}()$ clearly satisfies the latter two requirements to be completely modular. As the union of the existing orders, it maintains the relative ordering between existing properties. It is also clearly total. The first requirement of complete modularity, that composing well-formed orders creates an asymmetric and irreflexive order, requires an expanded definition of well-formedness for orders. Specifically, it prevents a module from adding a new dependency between existing properties, one that isn't induced by the composed ordering.

Definition 5.8 (Partially-ordered property set order well-formedness)**.** *In addition to the requirements in Definition 5.3, if a module M's property order $<^M$ using the partially-ordered property scheme relates property sets P_A and P_B ($P_A <^M_P P_B$) where both P_A and P_B are not introduced by M ($P_A \in S_A$, $S_A \in propertyCompose(M)$, $P_B \in S_B$, and*

*$S_B \in propertyCompose(M)$), then P_A and P_B must be related by the composed ordering $orderCompose_{PO}(M)$ (*i.e.*, if $orderCompose_{PO}(M)$ is $<$, $P_A <_P P_B$).*

Since preexisting properties cannot be newly related in any module, symmetry and reflexivity cannot be introduced in a composition. Thus $orderCompose_{PO}()$ is a completely modular ordering scheme when using the expanded definition of well-formedness for the property orders of modules.

Consider modules C and D from Figure 5.1 again. Under this definition of well-formedness of property orders, module C having $P_A <_P^C P_B$ is disallowed because P_A and P_B are imported but unrelated, being from independent modules. Module D introducing the ordering $P_B <_P^D P_A$ is also disallowed. Thus we cannot have the problem of C and D introducing conflicting orders for P_A and P_B that makes the composition for E impossible.

The potential difficulty with using partial orders lies in our inability to add new dependencies between preexisting properties. Module C cannot use P_A as a lemma in its modular proof of P_B. Such a dependency might be desirable if C defined the key relation of P_B using the key relation of P_A, as P_A might give information useful for proving P_B. Because these dependencies cannot be introduced, C either needs to find a different way to prove its properties, or it cannot introduce them. The same is true for D.

5.3 Ordering via Tags

Another completely modular ordering scheme relies on associating a tag with each property set, where each tag must be unique in the language library. Under this scheme, $\mathscr{P}$ for each module is a set of pairs of tags and sets of properties, with well-formed modules required to maintain the tags of imported properties (*e.g.*, if P is in a set associated with tag T in $\mathscr{P}^N$ and M builds on N, P must still be in a set associated with tag T in $\mathscr{P}^M$). Similar to names of syntax categories, constructors, and relations, the uniqueness of tags may be accomplished in practice by qualifying each tag with the name of the module introducing it.

Tags are drawn from a set with an asymmetric and irreflexive order $\prec$. The order for

property sets, $<^T$, is defined so $S <^T S'$ if and only if we have $T \prec T'$ for the tags T and T' associated with S and S' respectively. Each module uses this order as its own order, and order composition $orderCompose_T(M)$ is always this same order as well. The order $<^T$ is clearly a well-formed order, as the tag order is asymmetric and irreflexive, and it includes all relations in the composed order, as it and the composed order are the same. The composition scheme $orderCompose_T()$ is a completely modular ordering scheme, as it clearly satisfies the three criteria of being asymmetric and irreflexive, maintaining the same ordering as the modules it composes, and always existing.

If using this scheme, a module designer reads the properties his module imports and their tags, decides what lemmas his new properties need to use and which properties need his new properties as lemmas, and gives them tags that will satisfy the necessary ordering. When using tags, it is best to have a tag set and an ordering relation where there is always another tag available between two others. For example, using rational numbers as tags ensures a new property can always be added between two existing ones, as there will always be a tag that fits.

If our example language library from Figure 5.1 uses tags, module A assigns a tag T_A to the set containing its property P_A. Module B assigns a tag T_B to the set containing its property P_B. These tags then determine the ordering of the property sets. Since C and D both see the same tags, they both see the same order for the properties, and thus modules C, D, and E, all agree on the order.

This scheme also has the difficulty we saw with partially-ordered sets, that C and D may not have the ordering of P_A and P_B that they want. We can have $P_A <_P^T P_B$, $P_B <_P^T P_A$, or them being unrelated if the tag order $\prec$ that creates the property order is not total. Then either module C or D might have the order it wants based on the tags assigned to P_A and P_B when they are introduced, but not both. This is better for the module authors than the partially-ordered set scheme, as there is a chance the two properties are ordered the way they want, whereas there was no chance in the previous approach. However, at least one of C and D must find another way to write its modular proofs, as the shared order of P_A and P_B will not permit both of them to be used as lemmas in proving the other.

5.4 Not-Completely-Modular Ordering

All three criteria for completely modular ordering schemes are necessary to ensure all properties hold for *any* composition of modules. However, it may be that we sometimes do not care if any arbitrary set of modules can be composed, and would settle for limitations on which modules we may include in a composition in exchange for other benefits. Thus it might be that we would want to drop the totality criterion for ordering schemes, the one requiring a composition of well-formed orders always to exist. This criterion does not affect soundness, only guaranteed composability.

A more *laissez-faire* option than the previous two is to allow each module writer to build his own composed order from those of the modules on which his module builds. This order would need to obey the first two criteria from Definition 5.4, that a composed order is asymmetric and irreflexive, and that it respects the orders of the modules being composed by maintaining the relative order of all properties in the new order. Because Theorem 5.6 relies on only the second criterion directly, and the first through the requirement that a well-formed property order for a module is also asymmetric and irreflexive, an order built arbitrarily in this way would still produce valid proofs of all properties.

Arbitrarily ordering properties would solve the difficulty mentioned for the completely modular ordering schemes in both Sections 5.2 and 5.3, that modules may not be able to have the property orders they want even though those orders may be fine in the context of the module. Without restrictions on the order to maintain arbitrary composition, module C can create an order with $P_A <_P^C P_B$ as it wants, and module D can create an order with $P_B <_P^D P_A$ as it wants. Both modules can then prove their properties as they want. However, module E cannot make a well-formed order, as the orders from C and D are incompatible, and thus cannot have metatheory developed for it. Then, while C and D can both be written and prove all the properties they want in the way they want, they cannot be composed.

Under this approach, we cannot assume all properties from all modules included in a language composition are true for the composed language. We first must check an asymmetric,

irreflexive order for all the properties exists, one that respects all the property orders from modules included in the composition, that is, to compose the metatheory for $Lang(M)$, we must check a well-formed property order exists for the module M. If such an order exists, Theorem 5.6 tells us all the properties will hold.

Using this scheme for order composition makes the dummy module approach we used to compose our example language's independent extensions, seen in Section 2.3, possibly not succeed. Under a completely modular ordering scheme, a dummy module has no obligations for metatheory. It introduces nothing in the language nor any new properties, having only the imported properties but nothing new to write for modular proofs. It can then use the composed module ordering directly, giving the programmer creating a composition using a dummy module the language's metatheory completely free. When a composed ordering is not guaranteed to exist, a programmer using a dummy module for composition must try to create one to ensure the language's metatheory is what he expects. This can be accomplished using the $orderCompose_{PO}()$ ordering scheme and checking if the resulting order is asymmetric and irreflexive. Thus this scheme reduces, but does not completely eliminate, the work a programmer must do for composed metatheory.

It is clear there is no perfect scheme for composing property orders. Either we may find properties we want to use as lemmas are not available due to a quirk of composition but any modules will compose, as we see in using partially-ordered sets or tags, or we may find some modules do not compose but we can order lemmas in whatever way we want, as long as it does not contradict imported orders or break asymmetry or irreflexivity. It is not immediately clear whether one or the other of these is a more significant problem in practice, and thus which approach library originators should take. In our tools, discussed in the next chapter, we have chosen to use a tag ordering. However, more research into examples of the use of our reasoning framework, examples that contain large numbers of modules building on one another in interesting ways, is needed to identify how common each problem is and to give recommendations on which strategy to use.

Chapter 6

An Implementation of the Proof System

In addition to developing a framework for reasoning about languages, this thesis also examines the use of this framework in practice. To support the practical use of the reasoning framework for the purposes of this thesis and beyond, we provide implementations of both the language extensibility framework, allowing us to write specifications for extensible language modules as described in Chapter 2, and the reasoning framework, allowing us to declare and modularly prove metatheoretic properties about those specifications as described in Chapters 4 and 5.

These implementations comprise two separate systems. The Sterling system implements the language extensibility framework, permitting users to write module specifications and check they are well-formed. The Extensibella system then permits users to reason about such specifications, introducing metatheoretic properties and writing modular proofs of them. Additionally, both systems support the forms of composition relevant to them, with Sterling supporting the composition of module specifications to form full languages and Extensibella supporting the composition of modular proofs to form full, independently-checkable proofs of all properties included in a composition.

In this chapter we first look at how we specify language modules in Sterling and how it creates composed languages from module specifications in Section 6.1. We then look at declaring and writing modular proofs in Extensibella, followed by how Extensibella builds composed proofs for composed languages from the modular ones in Section 6.2.

6.1 Writing Extensible Languages in Sterling

The Sterling system implements our extensibility framework for defining languages from Chapter 2. Its design is inspired by the design of the Silver attribute grammar system [38] used for writing extensible languages with an extensibility framework similar to our own.

6.1.1 Language Specification

Sterling module specifications correspond quite directly to the language-specification 8-tuple $\langle \mathbb{B}, \mathscr{C}, \mathbb{C}, \mathscr{R}, \mathbb{R}, \mathscr{T}, \mathbb{T}, \mathbb{S} \rangle$ defining modules described in Chapter 2. Users specify the modules on which a new module builds ($\mathbb{B}$). They also declare new syntax categories ($\mathscr{C}$) and introduce constructors ($\mathbb{C}$) both for new syntax categories and ones from the modules on which they build. Specifications also include declarations of relations ($\mathscr{R}$), giving their argument types and marking which argument is the primary component, and rules ($\mathbb{R}$) defining both new and imported relations. Some rules are marked as default rules ($\mathbb{S}$). Finally, specifications also declare projection relations ($\mathscr{T}$) and rules defining them ($\mathbb{T}$).

The one part of a Sterling module specification that is not part of the 8-tuple is a module *name*, which we assume to be unique in the language library. These names are used in builds-on declarations to specify the modules on which a new one builds. They are also used for ensuring compositions do not encounter name problems. Since modules are assumed to be written independently, we could have, say, two constructors with the same name in a composition. To identify them uniquely, Sterling qualifies all names introduced in a module by the module name. For example, if the name of the host language module from Chapter 2 is H, the *add* constructor it introduces would actually have the name $H{:}add$, and the *vars* relation would actually have the name $H{:}vars$. Using fully-qualified names when writing rules would become quite tedious, so Sterling permits users to use short names when no other construct with the same short name is known. Thus we can write

$$\frac{vars(e_1, vr_1) \qquad vars(e_2, vr_2)}{vars(add(e_1, e_2), (vr_1 \cup vr_2))}\ \text{VR-Add}$$

and have Sterling treat it as if we had written

$$\frac{H{:}vars(e_1, vr_1) \qquad H{:}vars(e_2, vr_2)}{H{:}vars(H{:}add(e_1, e_2), (vr_1 \cup vr_2))}\ H{:}\text{VR-ADD}$$

giving us the benefits of unique names for composition but without the drudgery of fully writing them out each time we use them.

Once a Sterling module specification is written, the Sterling system checks it for well-formedness. To do this, Sterling reads the specification of the module and those of the ones on which it builds, gathering information about the declarations in each module. Once it knows all the declared constructs, it can proceed with checking the full set of modules is well-formed. One aspect of this check is that all the rules are well-typed, using relations and constructors with the types declared for them; this is no different from checking typing for a non-extensible system.

The more interesting parts of Sterling's checking of modules are for the well-formedness conditions in Section 2.2. One such condition is checking that a module does not build on itself. This could happen either by a direct declaration or transitively through another module, one on which it builds, also building on it. In gathering the declarations, Sterling checks it does not require a module it has already read, which would show a circularity in the module dependencies.

Most of the well-formedness checks relating to modularity depend on whether something is imported or new in a module. For example, Sterling checks that new rules defining imported relations have the primary component argument of the rule's conclusion built by a new constructor. In gathering declarations Sterling also tracks which module created each declaration. This allows it to check if a relation is imported and if the constructor in a rule's conclusion's primary component argument position is new. Sterling also checks each new relation with an imported primary component category has a default rule defined for it, and that each default rule has an unstructured primary component.

6.1.2 Language Composition

In addition to checking modules are well-formed, Sterling implements the composition into full languages described in Section 2.3. Most aspects of this composition are simply the union of the elements of the tuple to get full sets of syntax categories, syntax constructors, and relations for the full language. The interesting part of composition is in the instantiation of default rules for new constructors. Recall that a default rule from one module is instantiated for a constructor from another when the two modules are unrelated, that is, neither builds on the other. Sterling finds this set by finding all pairs of unrelated modules, then taking each new relation's default rule from one module and each new constructor building expressions in the relation's primary component category from the other to instantiate it. In our example language from Chapter 2, one pair of unrelated modules is the list module and the security module. Thus we take the default rule for the security module's *level* relation and instantiate it for each of the list module's new expression constructors, *nil*, *cons*, *null*, *head*, and *tail*.

We have two types of composed specifications that Sterling can output for use with Extensibella, one for modular reasoning and one for proofs for full languages. The modular reasoning specification incorporates the generic modules used in reasoning in Chapter 4. Recall that we have two generic modules that we use in reasoning, defined relative to a module M and a relation R. The default rule generic module $\mathbb{I}(M, R)$ introduces a generic constructor ι, and the proxy rule generic module $\mathbb{K}(M, R)$ introduces a generic constructor κ and the proxy rule for R instantiated for κ. These modules are used to write generic cases in proofs. Sterling creates the appropriate constructors and rules for the generic modules we might need in reasoning in the context of a specific module and adds them and any appropriate instantiations of default rules for them to the other rules it has for a composition.

The specification Sterling writes combines all the generic extensions we might need for proving any property the module might introduce. It does this since there is only one specification for reasoning in Extensibella, and we might need any of these generic extensions in

reasoning. Thus the language present for proving properties is actually larger than the one for which our theory calls. For example, in the context of the security extension S, we introduce and prove Property 3.3 that secure programs do not leak information. Our proof of this property, per Definition 4.6, is to be written for a language including the host language H, the security extension S, and the proxy rule generic module relative to the security extension and the property's key relation of statement evaluation, $\mathbb{K}(M, \Downarrow)$. However, the specification Sterling writes, and thus the one present for reasoning in Extensibella, will also include the proxy rule generic module for the statement typing relation, $\mathbb{K}(M, \mathit{type}_S)$. Extensibella will handle the inclusion of these extra generic modules in the language specification used for reasoning so each property is proven as if it were using exactly the language expected.

In order to generate the proxy rule generic modules for the specification, Sterling needs to know the proxy rules for relations, the set $\mathbb{Q}$ in the metatheory description tuple. Because they are needed to generate the language specification for reasoning, we include their declarations in Sterling along with the language specification, even though they conceptually belong to the metatheory description. Module writers can declare proxy rules for each new relation a module introduces. These are then used to produce the composed specification, being instantiated for the generic constructor when they might be needed for reasoning. If a relation is not given a proxy rule in the module specification, Sterling automatically generates one of the form

$$\frac{}{R(\overline{x})}$$

Having a proxy rule for every relation simplifies the implementation of Extensibella, and such a rule, if used to build the proxy version of the relation, does not impose any requirements on extension writers creating new rules.

The other type of composed specification Sterling can write for Extensibella to use is for full language proofs. These contain only the elements in the language of the module being composed as defined in Section 2.3, not anything from generic modules. When Extensibella

builds a composed set of proofs, it uses the full language specification instead of the modular reasoning one.

6.2 Modular Metatheory in Extensibella

In our reasoning framework, modules write proofs of metatheoretic properties in the logic $\mathcal{G}$ under restrictions on exactly how they may use its proof rules, with these modular proofs then used to construct composed proofs for composed languages. To implement this framework, we use the Abella proof assistant [2] that constructs proofs in $\mathcal{G}$. Specifically, Extensibella is a wrapper around Abella, taking proof commands from the user and checking their modular validity under our framework's restrictions, then passing them to Abella to apply the reasoning steps. Extensibella's implementation of proof composition takes the modular proofs and uses them to create Abella proofs of each metatheoretic property expected of a language composition. The result is an Abella development with declarations for the composed language's syntax and semantics and proofs of all its properties. This Abella development can be checked independently from our extensibility tools, demonstrating that all expected properties hold for the composed language.

Because Extensibella is a wrapper around Abella, its reasoning style is closely based on that of Abella. Thus we introduce Abella's reasoning style before discussing how we have implemented our reasoning framework, both the construction of modular proofs and proof composition, in Extensibella.

6.2.1 Introduction to Abella Reasoning

Abella reasoning developments consist of declaring theorems and proving them. Each theorem is a formula declared with a name. For example, the property that optimizing an expression does not change its evaluation result, Property 3.2, would be declared as

$$\texttt{Theorem}\ \text{opt\_expr\_correct} : \forall e, e', \gamma, v. opt_e(e, e') \supset \gamma \vdash e \Downarrow v \supset \gamma \vdash e' \Downarrow v$$

Once a theorem is declared, the user gives a proof of a sequent with the formula as the conclusion and no hypotheses. For Property 3.2, this would be

$$\varnothing : \varnothing \longrightarrow \forall e, e', \gamma, v. opt_e(e, e') \supset \gamma \vdash e \Downarrow v \supset \gamma \vdash e' \Downarrow v$$

After the proof is completed, the formula may be used as a lemma for proving future theorems, referring to it by name.

In proving theorems, Abella does not construct $\mathcal{G}$ proofs directly using the proof rules presented in Chapter 3. Rather, it uses *tactics*, commands that correspond to applying one or more proof rules. For example, the `search` tactic applies the $def\mathcal{R}$, $\exists\mathcal{R}$, $\wedge\mathcal{R}$, and $\vee\mathcal{R}$ rules, as well as the *id* rule and its annotated variants, to show the conclusion of the sequent being proven, applying as many of these rules as it takes to complete the proof. Tactics may also apply different proof rules depending on the hypotheses to which they are applied. As an example of a tactic that applies different proof rules in different situations, the `case` tactic applies the $def\mathcal{L}$ rule if it is applied to a hypothesis such as $opt_e(e, e')$, its annotated variants if applied to an annotated hypothesis, the $\vee\mathcal{L}$ rule if applied to a hypothesis of the form $F_1 \vee F_2$, and the $\wedge\mathcal{L}$ and $\exists\mathcal{L}$ rules if applied to hypotheses of the appropriate forms for them. Note that tactics may also include terms in them, a fact that will be relevant in building composed Abella proofs. For example, the `exists` tactic implements the $\exists\mathcal{R}$ rule more directly than `search`, allowing the user to specify a particular witness term for a conclusion formula $\exists x.F$.

Abella proofs are lists of tactics applying to the current sequent to prove in the proof state. The proof state is a list of sequents that need to be proven to complete the structured proof of the current theorem in $\mathcal{G}$ but have not yet been proven. Abella builds proofs through *proof search*, starting with the list being the sequent we want to end the $\mathcal{G}$ proof, then working backward to the leaves of the proof tree. In this process, each tactic removes the first sequent from the list, applies some proof rules to it that then require zero or more premise sequents to be proven, and adds those new premise sequents to the front of the list to be proven further. Once the list is empty, it means the full $\mathcal{G}$ proof has been constructed,

as all premise sequents of all proof rules used in it have also been given proofs, and the theorem is proven.

Consider proving Property 3.2 shown above, that optimizing expressions does not change their evaluation results. Initially our proof state is a single sequent

$$\varnothing : \varnothing \longrightarrow \forall e, e', \gamma, v. \mathit{opt}_e(e, e') \supset \gamma \vdash e \Downarrow v \supset \gamma \vdash e' \Downarrow v$$

We first apply the ind_1^1 rule using the `induction` tactic, giving a new single-sequent list

$$\varnothing : \mathit{IH} \longrightarrow \forall e, e', \gamma, v. \mathit{opt}_e(e, e')^{@} \supset \gamma \vdash e \Downarrow v \supset \gamma \vdash e' \Downarrow v$$

where we abbreviate the induction hypothesis as IH. We then apply the `intros` tactic that applies the $\forall\mathcal{R}$ and $\supset\mathcal{R}$ rules to the conclusion formula as long as they are applicable, getting a new singleton sequent list

$$e, e', \gamma, v : \mathit{IH}, \mathit{opt}_e(e, e')^{@}, \gamma \vdash e \Downarrow v \longrightarrow \gamma \vdash e' \Downarrow v$$

We follow this by analyzing $\mathit{opt}_e(e, e')^{@}$ with the `case` tactic, applying the $\mathit{def}\mathcal{L}^{@}$ rule, giving us a long list of sequents, one for each unifying rule:

$$\begin{gathered}
\gamma, v, e_1 : \mathit{IH}, \mathit{opt}_e(e_1, \mathit{false})^*, \gamma \vdash \mathit{not}(e_1) \Downarrow v \longrightarrow \gamma \vdash \mathit{true} \Downarrow v \\
\gamma, v, e_1 : \mathit{IH}, \mathit{opt}_e(e_1, \mathit{true})^*, \gamma \vdash \mathit{not}(e_1) \Downarrow v \longrightarrow \gamma \vdash \mathit{false} \Downarrow v \\
\gamma, v, e_1, e_1' : \mathit{IH}, \mathit{opt}_e(e_1, e_1')^*, \mathit{notBool}(e_1'), \gamma \vdash \mathit{not}(e_1) \Downarrow v \longrightarrow \gamma \vdash \mathit{not}(e_1') \Downarrow v \\
\vdots
\end{gathered}$$

For space reasons, we only show the first three here, for the OE-Not-T, OE-Not-F, and OE-Not-O rules, seen in Section A.4.2, handling the cases where the expression is constructed by *not* and the sub-expression optimizes to a constant *false*, constant *true*, and non-boolean-constant expression, respectively. The next tactic will apply to the first sequent, the one arising from the OE-Not-T rule. We can use the `case` tactic to analyze the evaluation derivation for $\mathit{not}(e_1)$, giving us two new sequents, one for if it was derived

using the E-Not-T rule and the other for if it was derived using the E-Not-F rule. Both are added to the start of the list of sequents, giving us

$$\gamma, e_1 : IH, opt_e(e_1, false)^*, \gamma \vdash e_1 \Downarrow false \longrightarrow \gamma \vdash true \Downarrow true$$
$$\gamma, e_1 : IH, opt_e(e_1, false)^*, \gamma \vdash e_1 \Downarrow true \longrightarrow \gamma \vdash true \Downarrow false$$
$$\gamma, v, e_1 : IH, opt_e(e_1, true)^*, \gamma \vdash not(e_1) \Downarrow v \longrightarrow \gamma \vdash false \Downarrow v$$
$$\gamma, v, e_1, e_1' : IH, opt_e(e_1, e_1')^*, notBool(e_1'), \gamma \vdash not(e_1) \Downarrow v \longrightarrow \gamma \vdash not(e_1') \Downarrow v$$
$$\vdots$$

The first sequent can be proven by the $def\mathcal{R}$ proof rule, which we can apply in Abella with the `search` tactic. This produces zero new sequents, so we move to the next sequent in the list. Here we can use the induction hypothesis to get a derivation of $\gamma \vdash false \Downarrow true$. To use the induction hypothesis, we use the `apply` tactic that combines the $\forall\mathcal{L}$ and $\supset\mathcal{L}$ rules, giving us a new sequent

$$\gamma, e_1 : IH, opt_e(e_1, false)^*, \gamma \vdash e_1 \Downarrow true, \gamma \vdash false \Downarrow true \longrightarrow \gamma \vdash true \Downarrow false$$

No language rules unify with evaluating *false* to *true* (*i.e.*, such an evaluation is impossible), so case analysis via the `case` tactic, implementing the $def\mathcal{L}$ proof rule, has no premise sequents, completing the proof of this sequent. That leaves us with the remainder of the list of sequents to prove:

$$\gamma, v, e_1 : IH, opt_e(e_1, true)^*, \gamma \vdash not(e_1) \Downarrow v \longrightarrow \gamma \vdash false \Downarrow v$$
$$\gamma, v, e_1, e_1' : IH, opt_e(e_1, e_1')^*, notBool(e_1'), \gamma \vdash not(e_1) \Downarrow v \longrightarrow \gamma \vdash not(e_1') \Downarrow v$$
$$\vdots$$

The proof continues, applying tactics, until the list of sequents is empty, showing the original formula has been proven.

Since the proof is a flat list of tactics and the remaining sequents are also a flat list, it appears the branching structure of the $\mathcal{G}$ proof is lost in the Abella proof. Not having the branching structure would be a problem for us in proof composition, as we need to be

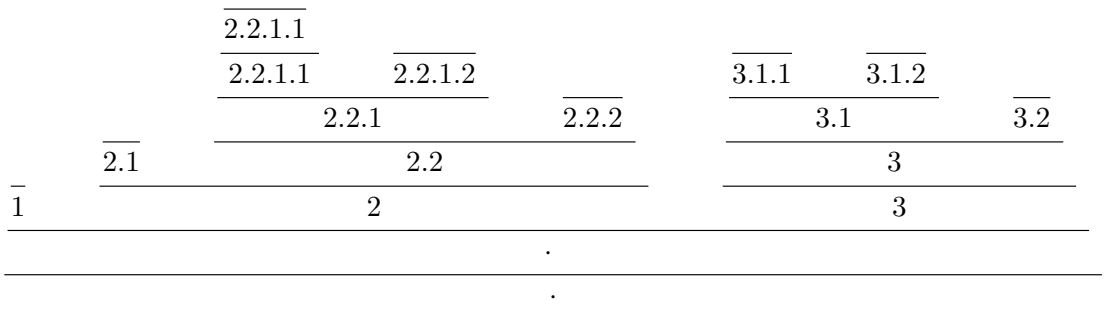


Figure 6.1: Demonstration of how subgoal numbers align with the proof structure

able to pull apart the modular proofs into the different top-level cases that are branches in the $\mathcal{G}$ proof. Fortunately, Abella maintains a record of the branching structure by pairing each sequent with a *subgoal number* that maintains a record of to which branch in a proof structure a particular sequent belongs. When writing proofs in Abella, seeing both the sequents to prove and their associated subgoal numbers helps track one's progress through the branches of the proof. Extensibella then uses the subgoal numbers Abella gives for the same purpose, bringing the structure it needs to the flat list of tactics.

Subgoal numbers are multi-part numbers, with a new part added to the number when a proof rule is applied that requires multiple premise sequents to be proven. For example, case analysis with the *def*$\mathcal{L}$ rule with multiple premise sequents adds a new part to the subgoal number, and the $\wedge\mathcal{R}$ rule that requires proving both conjuncts adds a new part to the subgoal number for each conjunct. The parts of the subgoal number are sequential, representing a path from the root of the proof through branching points. For example, the subgoal number 3.1.2 means a sequent belongs to the proof of the third premise sequent of the first branching point, the proof of the first premise sequent of the first branching point within that proof, and the proof of the second premise sequent of the first branching point within *that* proof. A proof starts with the special subgoal number $\cdot$, showing it has not had a branching point yet. Figure 6.1 shows a possible proof structure and the subgoal numbers that would be associated with each sequent in it, demonstrating how the subgoal numbers correspond to the proof structure. By combining the subgoal numbers for each

sequent with the tactics being applied to them, we can rebuild the $\mathcal{G}$ proof tree structure with its branching from the flat Abella proof.

Considering our start to a proof of Property 3.2 above, the initial sequent would be associated with subgoal number $\cdot$, as would the ones after the `induction` and `intros` tactics. Applying the `case` tactic to $opt_e(e, e')^{@}$ gives us a long list of new sequents. Associating them with their subgoal numbers, we get

$$\begin{array}{ll} 1: & \gamma, v, e_1 : IH, opt_e(e_1, false)^*, \gamma \vdash not(e_1) \Downarrow v \longrightarrow \gamma \vdash true \Downarrow v \\ 2: & \gamma, v, e_1 : IH, opt_e(e_1, true)^*, \gamma \vdash not(e_1) \Downarrow v \longrightarrow \gamma \vdash false \Downarrow v \\ 3: & \gamma, v, e_1, e_1' : IH, opt_e(e_1, e_1')^*, notBool(e_1'), \gamma \vdash not(e_1) \Downarrow v \longrightarrow \gamma \vdash not(e_1') \Downarrow v \\ & \vdots \end{array}$$

Applying the `case` tactic again, this time to $\gamma \vdash not(e_1) \Downarrow v$, gave us two new sequents as premises of the proof of the first sequent. Since the first sequent has subgoal number 1, the new premise sequents have subgoal numbers 1.1 and 1.2:

$$\begin{array}{ll} 1.1: & \gamma, e_1 : IH, opt_e(e_1, false)^*, \gamma \vdash e_1 \Downarrow false \longrightarrow \gamma \vdash true \Downarrow true \\ 1.2: & \gamma, e_1 : IH, opt_e(e_1, false)^*, \gamma \vdash e_1 \Downarrow true \longrightarrow \gamma \vdash true \Downarrow false \end{array}$$

We prove the first sequent with a single `search`, then prove the second by using `apply` with the induction hypothesis and analyzing the produced hypothesis with `case`. The sequent resulting from the use of `apply` also has subgoal number 1.2 because the `apply` does not require proving multiple premise sequents.

6.2.2 Modular Reasoning

In our reasoning framework, each module introduces a set of properties and an order of them to determine what properties may be used as lemmas in proving another in its metatheory description tuple, along with a set of proxy rules. A module both proves its own properties in restricted ways, possibly in the context of a language including some generic modules, and proves imported properties. These provide proofs of sequents that are sufficient to use to

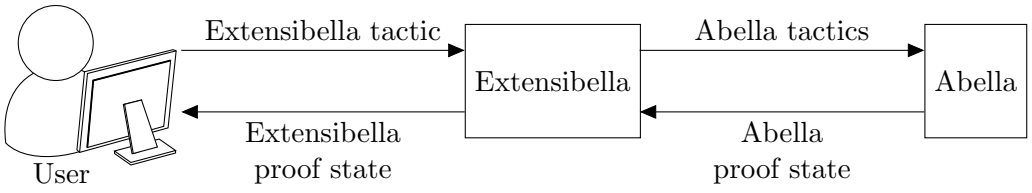


Figure 6.2: Workflow for communication between a user, Extensibella, and Abella

build a composed proof for any language. As mentioned earlier, our implementation requires the proxy rules to be declared by Sterling so it can create the specifications for modular reasoning. Extensibella implements the other portions of the metatheory description tuple and the proofs of the properties using Abella. The workflow for communication between a user, Extensibella, and Abella is shown in Figure 6.2. The user issues commands (*e.g.*, declaring the next property) and tactics to Extensibella, which it checks are valid according to its restrictions. Extensibella turns these into corresponding lists of commands and tactics for Abella. Abella processes the result of applying them and outputs the proof state, which Extensibella reads. It processes the sequents in the proof state, then presents the state to the user.

The first command a user gives to Extensibella for reasoning is a declaration of the language module about which a session will reason. Extensibella reads the specification Sterling wrote for modular reasoning, the one that includes any generic modules that could be needed for reasoning, and sends this specification to Abella as a set of $\mathcal{G}$ definitions. Because Abella doesn't have the type of language modularity our framework uses, the $\mathcal{G}$ definitions use fully-qualified names encoded into single names Abella can accept. However, Extensibella allows using unqualified names as Sterling does. Thus, whenever Extensibella reads a name from user input, it turns it into the intended fully-qualified name and encodes it to send to Abella. Similarly, when reading a proof state from Abella, which contains only fully-qualified names, Extensibella checks whether there are conflicts between the unqualified names, translating the conflict-free ones into the unqualified versions before presenting the proof state to the user, making it easier to read the proof state.

Consider proving Property 3.2 from the previous section, but now as a property in

Extensibella instead of a plain Abella theorem. The first sequent in the proof state we read back after the top-level case analysis is

$$\gamma, v, e_1 : (\forall e, e', \gamma, v. H{:}opt_e(e, e')^* \supset \gamma \vdash e\ (H{:}\Downarrow)\ v \supset \gamma \vdash e'\ (H{:}\Downarrow)\ v),$$
$$O{:}opt_e(e_1, H{:}false)^*, \gamma \vdash H{:}not(e_1)\ (H{:}\Downarrow)\ v \longrightarrow \gamma \vdash H{:}true\ (H{:}\Downarrow)\ v$$

As none of the unqualified names have conflicts, Extensibella displays the sequent to the user as

$$\gamma, v, e_1 : (\forall e, e', \gamma, v. opt_e(e, e')^* \supset \gamma \vdash e \Downarrow v \supset \gamma \vdash e' \Downarrow v),$$
$$opt_e(e_1, false)^*, \gamma \vdash not(e_1) \Downarrow v \longrightarrow \gamma \vdash true \Downarrow v$$

Because module names in actual developments are much longer than a single character, this becomes quite important in practice for making the proof state comprehensible.

Once we have our module specification, we can start declaring properties. In Extensibella, the user declares named properties with their formulas. As with elements from language specifications, Extensibella qualifies the names of properties, ensuring they are unique in a library. Property declarations are turned into `Theorem` declarations, with all names appropriately qualified, and sent to Abella. Extensibella property declarations also include information about the intended inductions for the canonical-form proof of the property so all proofs, both those given by the module introducing the property and those given by modules building on it, use the same form, ensuring their proofs will fit together in building a composition.

Recall from Chapter 4 that the canonical form of a property's proof is used by all modules to set up the top-level case analysis used in the modular proofs, and that it has the form of some uses of the ind^i_m rule at the end, with its premises being uses of the $\forall\mathcal{R}$ and $\supset\mathcal{R}$ rules, with the $def\mathcal{L}^{@^i}$ rule for the top-level case analysis for the property. In Abella, this turns into a list of tactics implementing the proof rules, specifically uses of the `induction`, `intros`, and `case` tactics.

When we have a mutually-inductive property set, the canonical form of the proof for the group is more complicated, as discussed in Section 4.5. It still ends with uses of the ind_m^i rule, but the premises of these are uses of the $\wedge\mathcal{R}$ rule to split the individual properties, then property-specific $\forall\mathcal{R}$, $\supset\mathcal{R}$, and $def\mathcal{L}^{@^i}$ rule uses for each property. This canonical form cannot be implemented simply as a list of tactics, since, in proving $P_1 \wedge P_2$, Abella expects the full proof of P_1 to be given before the proof of P_2 is started. Then we need to save the `intros` and `case` tactics that are specific to P_2 until after P_1 has been proven. Fortunately, the regularity of the canonical form means we know the subgoal number each property will have in its Abella proof, so we can recognize when the proof of one property is finished and the proof of the next one starts. Thus we know we are starting the proof of P_2 when Abella tells us the next sequent has subgoal number 2, and issue the canonical form's tactics then to carry out P_2's top-level case analysis.

Once the tactics for the canonical form of a property's proof have been passed to Abella, its proof state has a list of the sequents for the top-level cases. Each sequent corresponds to a rule Abella knows that has a conclusion unifying with the key relation. Extensibella then provides proofs for these sequents by reading tactics from the user and sending them to Abella. However, tactics read from the user cannot be sent straight to Abella. Our reasoning framework imposes restrictions, specified in Definitions 4.1 and 4.6, on using some proof rules, and Extensibella must check these restrictions are obeyed.

One such restriction is on case analysis with the $def\mathcal{L}$ rule and its annotated variants. Analyzing a hypothesis is allowed if its primary component is built by a constructor, if the primary component category is not extensible so no new rules can be added, or if the primary component is a generic constructor and the relation must be defined for it by instantiating the relation's default rule. In Abella, the $def\mathcal{L}$ rule and its variants are implemented by the `case` tactic. Extensibella checks each use of the `case` tactic to ensure the hypothesis it attempts to analyze can be analyzed without contravening these restrictions. In reading the specification Sterling wrote, Extensibella gathers information about the relations and their primary components, as well as the builds-on structure of the set of modules known. This lets it determine which argument to an atomic hypothesis is its primary component

to determine if it is built by a constructor. For example, in reading the specification for reasoning about the optimization module, it learns the primary component argument of $opt_e(e, e')$ is the first one and the primary component argument of $\gamma \vdash e \Downarrow v$ is the second one. Thus it can tell case analysis is allowable on $\gamma \vdash not(e_1) \Downarrow v$ but is not allowable on $opt_e(e_1, e_1')^*$.

Modular proofs are also not allowed to use the *def*$\mathcal{R}$ proof rule with language rules introduced by any of the proxy rule generic modules $\mathbb{K}(M, R)$. Recall from above that the *def*$\mathcal{R}$ rule is implemented in Abella as part of the `search` tactic that also implements several other proof rules, including the *id* rule and its annotated variants. We encode this ban into the language specification written by Sterling. When instantiating a proxy rule for a generic constructor κ as part of a proxy rule generic module, Sterling adds an extra false premise to the rule. Thus the proxy rule for statement evaluation, introduced as

$$\frac{proj_s(s, s') \qquad (\gamma,\ s') \Downarrow \gamma''}{(\gamma,\ s) \Downarrow \gamma'}\ \text{X-Q}$$

is instantiated for κ in Sterling as

$$\frac{proj_s(\kappa, s') \qquad (\gamma,\ s') \Downarrow \gamma'' \qquad \bot}{(\gamma,\ \kappa) \Downarrow \gamma'}\ \text{X-Q}(\kappa)$$

This extra premise of the rule prevents using it with *def*$\mathcal{R}$ in the proof unless $\bot$ is provable, in which case the proof can be completed with the $\bot\mathcal{L}$ rule rather than the disallowed use of the *def*$\mathcal{R}$ rule.

While this approach solves the problem of using the generic-instantiated proxy rule with the *def*$\mathcal{R}$ proof rule, it introduces another one. The top-level case analysis for a property will produce a case for this rule that will include $\bot$ as a hypothesis, making the $\bot\mathcal{L}$ rule applicable when it should not be. Consider the security property guaranteeing programs

passing the security extension's analysis do not leak information, Property 3.3:

$$\forall s, \Sigma, sl, \Sigma', \gamma_1, \gamma_1', \gamma_2, \gamma_2'.(\gamma_1,\ s) \Downarrow \gamma_1' \supset (\gamma_2,\ s) \Downarrow \gamma_2' \supset \Sigma\ sl \vdash secure(s, \Sigma') \supset$$
$$eqpublicvals(\Sigma, \gamma_1, \gamma_2) \supset eqpublicvals(\Sigma, \gamma_1', \gamma_2')$$

The top-level case analysis on the key relation for this property produces a sequent for the X-Q(κ) rule:

$$\Sigma, sl, \Sigma', \gamma_1, \gamma_1', \gamma_2, \gamma_2', s', \gamma'' : IH, (\gamma_2,\ \kappa) \Downarrow \gamma_2', \Sigma\ sl \vdash secure(\kappa, \Sigma'), eqpublicvals(\Sigma, \gamma_1, \gamma_2)$$
$$proj_s(\kappa, s'), (\gamma_1,\ s') \Downarrow \gamma'', \bot \longrightarrow eqpublicvals(\Sigma, \gamma_1', \gamma_2')$$

This sequent would be the correct one if it did not include the $\bot$ assumption. In reading a specification, Extensibella gathers information on the rules defining the relations, so it knows which one is the instantiated proxy rule that will have this extra, unwanted premise. From this, it can determine which subgoal number will be associated with the sequent and, just as tactics that are conceptually part of the canonical form are saved to be issued when certain subgoal numbers are reached, Extensibella can save a `clear` tactic, a tactic to remove hypotheses the user no longer wants, to remove the $\bot$ assumption before the user writes the generic proof. In proving the security property, Extensibella determines the generic case will be subgoal number 10, and thus when the proof reaches subgoal number 10 it issues a command to clear the $\bot$ assumption. This leaves us with only the hypotheses our theory expects, and thus a proof that will be valid for use in the composition.

Abella reasons about a single language specification in each development. It is for this reason that Sterling creates a single language specification including all the generic modules we might need, as it guarantees each modular proof has all the cases that will be needed for building a composed proof. Recall the proxy rule generic module relative to a module M and a relation R, $\mathbb{K}(M, R)$, introduces a new generic constructor κ that is specific to the relation R. Then a Sterling-created language specification often contains multiple different such constructors for different relations. The specification for the optimization extension

includes six such generic constructors, one each for expression typing, statement typing, the *vars* relation for the variables in an expression, the *value* predicate, expression evaluation, and statement evaluation. Each generic constructor has default rules from other modules instantiated for it as appropriate by the definition of language composition, so specifications often contain more rules for relations, and thus more cases from the top-level case analysis for a property, than are needed to fulfill the module's proof obligations. We can see this with Property 3.2, that optimizing an expression does not change its evaluation result, as, in addition to the rules for optimizing an expression in $\mathbb{R}^O$ and the default rule instantiated for ι that the reasoning framework expects to be proven, Abella also presents cases for the default rule instantiated for the generic constructors from the proxy rule generic modules for *vars*, *value*, and expression typing and evaluation. Just as Extensibella can determine with which subgoal number a needed generic case for an instantiated proxy rule will be associated, so it can determine which subgoal number will be associated with unneeded cases. It can then issue a `skip` tactic, an unsound Abella tactic that treats a sequent as solved to ease proof exploration, when that subgoal is reached. This means the user needs to prove only the cases that would be part of a proof in the language specified by Definition 4.1 or Definition 4.6, meaning Extensibella's use of `skip` is sound for modular reasoning purposes. Note that these extra cases, which Extensibella clears before the user sees them, are the only ways the extra constructs could affect proofs. As explained in Section 4.5, having extra generic constructors and rules cannot affect the proofs of needed cases. Then the potentially-over-sized Sterling specification for modular reasoning does not have an impact on what the user actually needs to prove in Extensibella.

In reasoning, a user declares the properties he wants in the order he wants them, this order determining which properties may be used as lemmas in proving others. Once a module's full set of properties and proofs are written, the user compiles the reasoning module to allow other reasoning modules to build on it. This compilation assigns each property a tag that is used for a tag-based ordering scheme for composing property orders, as described in Section 5.3, creating the order in the metatheory description tuple. The tags Extensibella uses are pairs of a rational number and the module name, making them

unique across all modules. For example, our host language's type preservation property might be tagged as $(\frac{15}{2}, H)$.

Recall that extension modules have obligations for extending the proofs of properties introduced by the modules on which they build. Modules building on others determine the property order for the imported properties by the tags assigned in compilation. Extensions additionally may introduce their own new properties, so, when reasoning about an extension module's specification, an Extensibella user both introduces new properties and declares when he is ready to write an extended proof for an imported one. Extensibella checks the declarations for extending imported properties come in the order their tags specify. The full set of proofs for an extension module is only accepted as complete once all the imported properties have had their proofs extended as needed. For example, the security extension from Chapter 2 must fulfill its obligations in proving the host language's properties, such as the statement version of type preservation, for its new *secdecl* construct in addition to proving its new properties for its Extensibella development to be complete.

Once all the required modular proofs have been written, the extension reasoning module, too, can be compiled, to allow other modules to build on it. Imported properties are assigned the same tags they already had, but new properties are assigned tags corresponding to their placement relative to the existing properties. For example, a module M might import properties with tags $(1, H)$ and $(3, H)$. It then might introduce a new property ordered between them. The Extensibella compilation of the new module then assigns the tags $(1, H)$ and $(3, H)$ to the imported properties as in the imported module, but it also generates a new tag for the new property. This tag must fall between $(1, H)$ and $(3, H)$, so it may be the new property is given the tag $(2, M)$, reflecting the intended order of the three properties.

6.2.3 Proof Composition

Recall from Section 4.4 that proof composition involves taking the $\mathcal{G}$ proofs of sequents written as part of modular reasoning and using them to build a complete $\mathcal{G}$ proof of a property for a full language. Implementing this in Extensibella means taking the underlying Abella proofs of sequents from the Extensibella proofs for each module and using them

to build a complete Abella proof of each property for a full language. Moreover, as per Chapter 5, we want to create a full Abella development, one that has a specification of the language and proofs of all the metatheoretic properties from all the modules. This can be accomplished by writing the composed proofs of the properties in the order specified by their tags, ensuring all properties needed as lemmas are proven before being used. The produced proofs can be checked by Abella independently of our extensible tools, demonstrating the composed language does have the expected properties.

We build the composed proof for a property as a canonical-form proof for it, starting with the same ind_m^i, $\forall\mathcal{R}$, $\supset\mathcal{R}$, and $def\mathcal{L}^{@^i}$ rules as were used to set up the proof written by each module. As discussed in the prior section, this canonical-form proof structure turns into a list of Abella tactics, giving us a list of sequents to prove, the same sequents that are expected as premises of the $def\mathcal{L}^{@^i}$ rule from the $\mathcal{G}$ canonical-form proof. The main work in the composition is supplying the proofs of these sequents arising from the top-level case analysis using the modular proofs.

The first problem to solve in this respect is identifying the proofs of different cases from the modular proofs. Whereas the modular $\mathcal{G}$ proofs used in building composed proofs in Section 4.4 have the branching structure built into them, the modular proofs written by Extensibella are flat lists of tactics to be applied, which do not tell us when one case is done and the next begins. However, as discussed in Section 6.2.1, the subgoal numbers that were associated with the sequents being proven by each tactic let us rebuild the structure, showing when each case begins and ends. Then in proving Property 3.2, where our top-level case analysis produces the proof state

$$\begin{array}{ll} 1: & \gamma, v, e_1 : IH, opt_e(e_1, false)^*, \gamma \vdash not(e_1) \Downarrow v \longrightarrow \gamma \vdash true \Downarrow v \\ 2: & \gamma, v, e_1 : IH, opt_e(e_1, true)^*, \gamma \vdash not(e_1) \Downarrow v \longrightarrow \gamma \vdash false \Downarrow v \\ 3: & \gamma, v, e_1, e_1' : IH, opt_e(e_1, e_1')^*, notBool(e_1'), \gamma \vdash not(e_1) \Downarrow v \longrightarrow \gamma \vdash not(e_1') \Downarrow v \\ & \vdots \end{array}$$

we know that any tactics proving a sequent associated with a subgoal number starting with 1 (*e.g.*, 1.1, 1.2, 1.2.1) are associated with the first case in the proof. The Extensibella

composition process thus starts by taking each modular proof and annotating each tactic command in it with the first subgoal number and sequent in the Abella proof state in checking the proof. It then builds the composed proof using these annotated tactic lists, where the subgoal numbers provide the structure needed for the composition.

In Section 4.4, we split the proof of the existence of a composed proof into two parts, one for when the composed proof did not need to use the proxy version of its key relation (Theorem 4.24) and one when it did (Theorem 4.25), which depend on whether the key relation and its primary component category were introduced in the same module or not. These correspond, respectively, to scenarios A and B and scenarios C and D in Figure 4.1. We do the same here in discussing Extensibella's proof composition, considering first the case where we do not need to use the proxy version.

Compositions Not Using the Proxy Version of the Key Relation

If we are using the key relation directly in the composed proof, rather than its proxy version, it must be that the property and key relation are introduced by the same module. Then the only types of rules defining the key relation that can be in the composed language are those from the module introducing the property (known rules in the terminology of Section 4.1) and modules building on it (new rules in the terminology of Section 4.1), and instantiations of the default rule. The former two categories are rules for which some module included in the composition wrote a proof directly. In Lemma 4.9, we showed the $\mathcal{G}$ proofs of such cases can be used directly in the composition proof. The list of Abella tactics corresponding to the Extensibella proof can similarly be used directly in the composed proof, as they will apply the same $\mathcal{G}$ proof steps as in the modular proof.

The latter category, instantiated default rules, are more complex. As in Lemma 4.13, the generic constructor ι has been replaced by a term built by a new constructor from another extension. While the structure of the proof for the composed language stays the same as the one used for the generic proof, there are two changes we need Extensibella to make to it. First, we need to replace any occurrences of the generic constructor ι in the proof, such as witnesses with the `exists` tactic, with the new term for which the

default rule was instantiated. Second, as the new term replacing ι has meta-variables as its arguments, it may be that Abella generates different names for existing meta-variables as we move through the proof than it did in the modular context. To solve both of these problems, Extensibella gives the commands for the composed proof to Abella as it is built so it has access to the proof state for the composition. It then maps the first sequent in the proof state for the composed proof onto the first sequent in the proof state for the generic proof. This mapping produces a substitution for variables in the generic proof's sequent, mapping them to the corresponding variable names in the composed proof's sequent, as well as mapping the generic constructor ι to the new term replacing it. For example, the sequent for the generic proof for Property 3.2 is

$$\gamma, v : \forall e, e', \gamma, v. opt_e(e, e')^{@} \supset \gamma \vdash e \Downarrow v \supset \gamma \vdash e' \Downarrow v, \gamma \vdash \iota \Downarrow v \longrightarrow \gamma \vdash \iota \Downarrow v$$

The sequent for the composed proof for the case where the default rule is instantiated for the list extension's *cons* constructor is

$$\gamma, v, e_1, e_2 : \forall e, e', \gamma, v. opt_e(e, e')^{@} \supset \gamma \vdash e \Downarrow v \supset \gamma \vdash e' \Downarrow v, \\ \gamma \vdash cons(e_1, e_2) \Downarrow v \longrightarrow \gamma \vdash cons(e_1, e_2) \Downarrow v$$

Because the variable names e_1 and e_2 for the new term do not conflict with existing ones, the mapping of the old variable names to the new ones is identity. We also have a mapping of the generic term ι to the new term $cons(e_1, e_2)$. With such a mapping in hand, Extensibella can modify each tactic in the list-of-tactics proof for the generic case to fit each case for an instantiation of the default rule that occurs in the composition. For each case in a composed proof, Extensibella either applies the corresponding modular proof directly or the appropriately-modified generic one, thus completing the proof.

Compositions Using the Proxy Version of the Key Relation

If the property and its key relation were introduced in separate modules, we need to use the proxy version of the key relation to create the proof for the composed language. As in Theorem 4.25, Extensibella splits the proof into two parts, providing a proof of

$$\forall \overline{x}.R_P(\overline{t}) \supset F$$

using the proofs written by each module, then using this as a lemma along with $addP(R)$ to prove the actual property

$$\forall \overline{x}.R(\overline{t}) \supset F$$

The latter part is simple and straight-forward, so we discuss only the former part here.

Such a property potentially has four types of cases that arise in a composed proof. There may be cases from the same three categories as we saw with properties not using the proxy versions of their key relations. The fourth type of case, and the one that is new here, is for rules introduced independently of the property by modules that do not know the property (independent rules in the terminology of Section 4.1), and thus do not provide proofs for it for their rules. We consider the first two types of cases together, then look at the third.

For those cases we saw when not using the proxy version of the key relation, we have the same considerations as there, plus handling the change of formulas of the form $R(\overline{t})^{*^i}$ into formulas of the form $R_P(\overline{t})^{*^i}$, and the same change for those annotated with $@^i$. In Lemma 4.17, we saw the essential change here was in the annotated versions of the *id* rule, where the modular proof might have used a hypothesis of the form $R(\overline{t})^{*^i}$ or $R(\overline{t})^{@^i}$ to prove an un-annotated conclusion. In the composition, these hypotheses now have the form $R_P(\overline{t})^{*^i}$ and $R_P(\overline{t})^{@^i}$. The solution in the direct $\mathcal{G}$ proofs was to apply the $dropP(R)$ property whenever we used such a rule. Extensibella uses the same idea, but has to do so somewhat indirectly. Abella, and therefore Extensibella, implements the *id* rule and its annotated variants as part of the `search` tactic, so Extensibella does not know which premises using the proxy version of the key relation, if any, are being used with an annotated

version of the *id* rule. However, Extensibella knows the `search` tactic might need the non-proxy-version of any hypotheses using the proxy version and it knows what those hypotheses are. Thus, in building the composition, Extensibella proactively applies the $dropP(R)$ property to all hypotheses that might need it, adding this to the steps for using the proofs written by modules discussed for properties that don't use the proxy version of the key relation.

Consider the case for the X-SEQ rule in proving Property 3.3, that secure programs do not leak private information. The proof proceeds by analyzing the second evaluation that uses γ_2, then by analyzing the derivation of $\Sigma\ sl \vdash secure(seq(s_1, s_2), \Sigma')$, and applying the induction hypothesis to the evaluations of s_1 and s_2. In the modular proof given by the security extension, this leaves us with a sequent

$$\begin{array}{l}\Sigma, sl, \Sigma', \gamma_1, \gamma_1', \gamma_2, \gamma_2', s_1, s_2, \gamma_1'', \gamma_2'', \Sigma'' : IH, eqpublicvals(\Sigma, \gamma_1, \gamma_2), \\ \qquad ((\gamma_1,\ s_1) \Downarrow \gamma_1'')^*, ((\gamma_1'',\ s_2) \Downarrow \gamma_1')^*, (\gamma_2,\ s_1) \Downarrow \gamma_2'', (\gamma_2'',\ s_2) \Downarrow \gamma_2', \\ \qquad\qquad \Sigma\ sl \vdash secure(s_1, \Sigma''), \Sigma''\ sl \vdash secure(s_2, \Sigma'), \\ \qquad\qquad eqpublicvals(\Sigma'', \gamma_1'', \gamma_2''), eqpublicvals(\Sigma', \gamma_1', \gamma_2') \\ \qquad\qquad\qquad \longrightarrow eqpublicvals(\Sigma', \gamma_1', \gamma_2')\end{array}$$

where *IH* is the induction hypothesis. At this point, the proof can be completed by the `search` tactic, since the conclusion is exactly the same as one of the hypotheses. In building the composed proof, we can apply the same line of reasoning, bringing us to a sequent

$$\begin{array}{l}\Sigma, sl, \Sigma', \gamma_1, \gamma_1', \gamma_2, \gamma_2', s_1, s_2, \gamma_1'', \gamma_2'', \Sigma'' : IH_P, eqpublicvals(\Sigma, \gamma_1, \gamma_2), \\ \qquad ((\gamma_1,\ s_1)\ \Downarrow_P\ \gamma_1'')^*, ((\gamma_1'',\ s_2)\ \Downarrow_P\ \gamma_1')^*, (\gamma_2,\ s_1) \Downarrow \gamma_2'', (\gamma_2'',\ s_2) \Downarrow \gamma_2', \\ \qquad\qquad \Sigma\ sl \vdash secure(s_1, \Sigma''), \Sigma''\ sl \vdash secure(s_2, \Sigma'), \\ \qquad\qquad eqpublicvals(\Sigma'', \gamma_1'', \gamma_2''), eqpublicvals(\Sigma', \gamma_1', \gamma_2') \\ \qquad\qquad\qquad \longrightarrow eqpublicvals(\Sigma', \gamma_1', \gamma_2')\end{array}$$

where IH_P is the induction hypothesis using the proxy version of the key relation. Before applying the `search` tactic, Extensibella sees the two evaluation derivations using the proxy version and uses the $dropP(eval)$ property to produce hypotheses $(\gamma_1,\ s_1) \Downarrow \gamma_1''$ and $(\gamma_1'',\ s_2) \Downarrow \gamma_1'$ before using the `search` tactic. Note this is not necessary in this case, as those hypotheses cannot be used to prove the conclusion, but Extensibella applies $dropP(R)$ liberally to be conservative with respect to its necessity.

The last remaining piece of the proof composition is the handling of the proof cases for rules introduced by modules that don't know the property exists. The rule for each of these cases in the composition combines the actual rule given by the module and the proxy rule given by the module introducing the relation. For example, in the composed proof of Property 3.3, there is a case for the X_P-Splitlist rule:

$$\frac{\begin{array}{c}\gamma \vdash e \Downarrow cons(v_1, v_2) \qquad n_{hd} \neq n_{tl} \qquad update(\gamma, n_{hd}, v_1, \gamma') \\ update(\gamma', n_{tl}, v_2, \gamma'') \qquad proj_s(splitlist(n_{hd}, n_{tl}, e), s') \qquad (\gamma,\ s')\ \Downarrow_P\ \gamma'''\end{array}}{(\gamma,\ splitlist(n_{hd}, n_{tl}, e))\ \Downarrow_P\ \gamma''}\ X_P\text{-Splitlist}$$

This combines the premises of the X-Splitlist rule, the first four premises, with the premises of the X-Q proxy rule for statement evaluation, the projection and evaluation of the projection. The intuition behind proving such cases is that the module introducing the property provided a generic case for the proxy rule instantiated for a generic constructor κ. Since the sequent in the composed language includes the premises of the proxy rule, we can use the proof of the generic case to create a proof of the composed cases for the new rules.

To use the generic proof for the new cases in the composition, we need to map the generic proof state's first sequent to the first sequent in the composition's proof state. This is similar to the mapping we saw above for mapping the generic case for the default rule instantiated with ι to the composed proof's sequent. However, this mapping maps the variables in the original to terms, which could be variables or structured terms; maps the generic κ constructor to a term instead of mapping ι to a term; and it must account for the change from the key relation to its proxy version and added hypotheses in the new sequent

Generic proof sequent:	Composed proof sequent for X_P-SPLITLIST
$\Sigma, sl, \Sigma', \gamma_1, \gamma_1', \gamma_2, \gamma_2', s, :$	$\Sigma, sl, \Sigma', \gamma_1, \gamma_1', \gamma_2, \gamma_2', s, , n_{hd}, n_{tl}, e, v_1, v_2, \gamma' :$
$IH,$	$IH_P,$
$(\gamma_2,\ \kappa) \Downarrow \gamma_2',$	$(\gamma_2,\ splitlist(n_{hd}, n_{tl}, e)) \Downarrow \gamma_2',$
$\Sigma\ sl \vdash secure(\kappa, \Sigma'),$	$\Sigma\ sl \vdash secure(splitlist(n_{hd}, n_{tl}, e), \Sigma'),$
$eqpublicvals(\Sigma, \gamma_1, \gamma_2),$	$eqpublicvals(\Sigma, \gamma_1, \gamma_2),$
	$\gamma_1 \vdash e \Downarrow cons(v_1, v_2),$
	$n_{hd} \neq n_{tl},$
	$update(\gamma_1, n_{hd}, v_1, \gamma'),$
	$update(\gamma', n_{tl}, v_2, \gamma_1'),$
$proj_s(\kappa, s'),$	$proj_s(splitlist(n_{hd}, n_{tl}, e), s'),$
$(\gamma_1,\ s') \Downarrow \gamma_1'$	$(\gamma_1,\ s')\ \Downarrow_P\ \gamma_1'$
$\longrightarrow eqpublicvals(\Sigma', \gamma_1', \gamma_2')$	$\longrightarrow eqpublicvals(\Sigma', \gamma_1', \gamma_2')$

Figure 6.3: Mapping between hypotheses in a generic proof's sequent and a corresponding sequent in a composed proof

that were not present in the original. Figure 6.3 shows how the sequent from the generic proof (left) maps to the sequent in the composed proof for X_P-SPLITLIST (right) by aligning the mapped hypotheses. Note the four hypotheses corresponding to the premises of the X-SPLITLIST rule do not have hypotheses from the sequent for the generic proof mapping to them, as they are specific to the list extension's *splitlist* construct. What is required of the mapping is that each hypothesis in the sequent for the generic proof correspond to one in the sequent for the composed proof, as these are the hypotheses that may be used in the generic proof, and thus that may be used in the composed proof. Extensibella uses this mapping in building the composed proof as it did for the generic proof written for the default rule instantiated for ι, replacing the variables from the original proof with the terms to which they map in the new sequent and replacing the generic constructor κ with the term to which it maps.

The one remaining issue in using generic proofs written for the proxy rule instantiated for the generic constructor κ for the rules written by other extensions is that certain branches of the proof may be pruned by replacing variables in the original proof with structured terms; this was noted also in the proof of Theorem 3.4. Extensibella can recognize that a case from the original proof has been pruned by checking if the generic proof's sequent for

the branch maps to the composed proof's sequent. If it does not map, the branch has been pruned, and the rest of the tactics in the branch, as determined by the subgoal numbers with which they are associated, can be dropped. If it does map, the branch is still present in the composed proof, and its tactics can be applied, once appropriately substituted according to the mapping found. In this way, the generic proof can be used to complete the proof of a sequent in the composition.

Thus, by applying the correct portions of the modular proofs from each Extensibella development, we can complete the proof of each property for the composed language. While we have implicitly couched our discussion here in terms of proving individual properties, it applies to full sets of mutually-inductive properties as well. Then we can produce a full Abella development demonstrating all the properties hold for a composed language.

Chapter 7

Using the Framework in Practice

In this chapter, we examine the practical application of our framework, looking at how our proof restrictions and the module structure of specifications affect the proofs of properties, both in how we write the proofs and what properties are provable, as well as how modular metatheory influences language specifications themselves. We do this by looking at several example languages we have developed using our Sterling and Extensibella implementations of our extensibility and reasoning frameworks.

Overall, we have found our framework is quite effective, proving a number of properties for several different languages. These include properties such as progress and type preservation for the lambda calculus and type and evaluation uniqueness for imperative languages. The Sterling and Extensibella code for our examples can be found on our website [26]. The proofs of many metatheoretic properties naturally follow the restrictions on case analysis imposed by Definitions 4.1 and 4.6, so the approaches used in non-extensible settings can often be imported directly to the extensible one. We have found generic cases also often fit into the natural way of structuring the proofs of properties that would be used in a non-extensible setting.

Not all proofs work out as naturally in the extensible setting as in non-extensible settings, however, and we consider some such cases by example in the remainder of this chapter. First, extensibility does occasionally cause problems in writing proofs, either due to wanting to use a disallowed case analysis or wanting to introduce a property in an extension, but where we cannot prove the generic case. We consider how we overcome such problems in specific circumstances in Sections 7.1 and 7.2 as a model for how they might be overcome

in general. We further consider how extensibility affects language metatheory, specifically how it relates to introducing properties in extensions in the next two sections. Section 7.3 considers how relations and properties about them differ when introduced by extensions as opposed to being introduced by the module introducing the relation's primary component category. Section 7.4 then considers how choices the designer of the host language makes regarding constructs to include in the language and projection constraints to introduce affect the constructs and properties extensions may introduce. Section 7.5 closes by laying out considerations language designers should bear in mind when writing language specifications and property sets for modules, gleaned from our experiences in writing these and other examples, to help them in creating modules on which it is easy for others to build.

7.1 Overcoming Restrictions on Case Analysis

We start by considering a situation where our restrictions on case analysis from Definitions 4.1 and 4.6 prevent us from proving what we want in the most direct way. Suppose we want to prove equality is decidable for expressions in the language from Chapter 2. This property can be expressed by the formula

$$\forall e_1, e_2. e_1 = e_2 \vee (e_1 = e_2 \supset \bot)$$

Proving this statement depends on analyzing the structures of e_1 and e_2 to see if they are the same or not. We cannot do this in $\mathcal{G}$ directly as its types do not provide information about structure. In Abella, this limitation is commonly addressed by introducing *is relations*, relations that encode the structure of all constructors of a type. These are commonly given names starting with "is", hence the name of the class of relations. They are so common that Sterling automatically introduces such a relation for each syntax category in a specification. For expressions in our example language, we can have a relation is_e, some of the rules for which are shown in Figure 7.1. The formula for decidable equality then becomes

$$\forall e_1, e_2. is\_e(e_1) \supset is\_e(e_2) \supset e_1 = e_2 \vee (e_1 = e_2 \supset \bot)$$

$$\frac{}{is\_e(true)}\ \textsc{IsE-True} \qquad \frac{}{is\_e(false)}\ \textsc{IsE-False}$$

$$\frac{is\_n(n)}{is\_e(var(n))}\ \textsc{IsE-Var} \qquad \frac{is\_i(i)}{is\_e(intlit(i))}\ \textsc{IsE-Intlit}$$

$$\frac{is\_e(e_1) \qquad is\_e(e_2)}{is\_e(add(e_1, e_2))}\ \textsc{IsE-Add} \qquad \frac{is\_e(e_1) \qquad is\_e(e_2)}{is\_e(eq(e_1, e_2))}\ \textsc{IsE-Eq}$$

$$\frac{is\_e(e_1) \qquad is\_e(e_2)}{is\_e(gt(e_1, e_2))}\ \textsc{IsE-Gt} \qquad \frac{is\_e(e)}{is\_e(not(e))}\ \textsc{IsE-Not}$$

Figure 7.1: Rules in the host language for the is relation for expressions

This we can prove in $\mathcal{G}$, since we can analyze both $is\_e(e_1)$ and $is\_e(e_2)$ to get information about the expressions' structures, finding each is either built by the same constructor and their sub-expressions are either equal or not, or they are built by different constructors and thus are not equal.

This proof structure does not work in the extensible setting, however. Considering this as a property introduced by our host language, the derivation of the key relation in this property is $is\_e(e_1)$, so our top-level case analysis gives us cases for when e_1 is constructed by the *var* constructor, the *intlit* constructor, and so on. However, once we know the structure of e_1, we cannot then examine the structure of e_2 by analyzing $is\_e(e_2)$ because its primary component is unstructured; this is the very reason we want to analyze it, to give it structure.

To get around this restriction, we can prove separate properties to use as lemmas in the main proof that will allow us to specify whether or not e_2 has a particular form. For each constructor it introduces, the host language also introduces and proves a property stating an expression's top-level symbol is that constructor or not. For the *true* constructor, we have

$$\forall e.is\_e(e) \supset e = true \vee (e = true \supset \bot)$$

For the *add* constructor, we have

$$\forall e.is\_e(e) \supset (\exists e_1, e_2.e = add(e_1, e_2)) \vee ((\exists e_1, e_2.e = add(e_1, e_2)) \supset \bot)$$

These properties are easily provable in our framework.

The host language's modular proof that equality is decidable then applies the appropriate lemma in each case to check whether e_2 is built by the same constructor as e_1. For example, if e_1 is built by the *add* constructor, we apply the above property that states any expression is either built by the *add* constructor or not to $is\_e(e_2)$. If it is not built by the *add* constructor, e_1 and e_2 are not equal and we can easily finish the proof. If e_2 is built by the *add* constructor, e_2 is actually $add(e_1', e_2')$, and the derivation of $is\_e(e_2)$ is actually a derivation of $is\_e(add(e_1', e_2'))$. Since the primary component of this is now structured, we can analyze it with the $def\mathcal{L}$ rule to get $is\_e(e_1')$ and $is\_e(e_2')$ for use with the induction hypothesis with is_e derivations for the sub-expressions of e_1.

The list extension must also provide a new modular proof of each of these properties for the constructs it introduces. For the form lemmas introduced by the host language, such as that an expression is built by *add* or not, the proof for each new construct is simple, as the new constructors like *cons* cannot be equal to the constructors introduced by the host language. For decidable equality itself, we need to apply the same strategy as in the host language, creating new form lemmas for the list constructs, such as

$$\forall e.is\_e(e) \supset (\exists e_1, e_2.e = cons(e_1, e_2)) \vee ((\exists e_1, e_2.e = cons(e_1, e_2)) \supset \bot)$$

that we can use in the same manner as in the host language. As long as the host language proved $addP(is\_e)$, allowing us to use the host language's is_e relation as the key relation for extension-introduced properties, these extension-introduced form lemmas can be proven as easily as those introduced by the host language. The proofs for cases for known constructors are easy, and the generic constructor in the generic case is not equal to any known constructor, so the generic case is easily provable as well.

Thus we see that, while our restrictions on proofs prevent proving the decidability of equality directly as we would in a non-extensible setting, we can use auxiliary properties as lemmas to replace the disallowed second case analysis. This is a strategy that can similarly be applied in other situations, introducing lemmas to specify what we would use a second

case analysis to show in a non-extensible setting.

7.2 Simply-Typed Lambda Calculus

Our next example examines how a property might not be provable when introduced by a module other than the one introducing its primary component category and key relation because its generic case has no proof, and how we might overcome this difficulty in some cases. The language we consider here uses as its host language the simply-typed lambda calculus with numbers and addition.[1] This module introduces categories for terms and types, with the syntax shown in Figure 7.2. The host language's module also defines typing, small-step evaluation, and substitution of a term for a name, with rules as expected. Finally, it introduces a predicate for determining whether a term is a value. Rules for evaluation and identifying values are also shown in Figure 7.2.

Term projection is defined with an extra argument of a type context, allowing projections to be based on the types of terms. We have two extensions, both of which take advantage of this extra argument to use types in projection. The first extension introduces let bindings that project to applications of abstractions, with the type of the bound variable being the type of the term bound by the let. This is the P-Let rule in Figure 7.3. The other extension we write introduces syntax for pairs of terms of the same type, projecting the first and second elements out of pairs, and pair types. Pairs of terms of type τ project to functions taking a selector of type $\tau \to \tau \to \tau$ that chooses either the first or second element out of the pair, with the projections of the *fst* and *snd* selectors applying the pairs to such selectors. These rules are also shown in Figure 7.3. In addition to defining new syntax and projections for it, both of these extensions also define the relations from the host language for their new syntax, with rules as one would expect for the constructs. The only new construct identified as a value is the *pair* construct, which is a value if both elements of the pair are also values.

Our host language introduces a number of properties, most of which we will not discuss

[1] Sterling and Extensibella code available online at `https://mmel.cs.umn.edu/extensibella/examples/stlc/description.html`

$$\begin{array}{rcl@{\qquad}rcl} tm & ::= & \lambda x : \tau.tm & \tau & ::= & intTy \\ & | & tm\ tm & & | & \tau \to \tau \\ & | & x & & & \\ & | & tm + tm & & & \\ & | & int(i) & & & \end{array}$$

$\boxed{eval(tm^*, tm)}$

$$\frac{eval(t_1, t_1')}{eval(t_1\ t_2, t_1'\ t_2)}\ \text{E-App-Step1} \qquad \frac{value(t_1) \qquad eval(t_2, t_2')}{eval(t_1\ t_2, t_1\ t_2')}\ \text{E-App-Step2}$$

$$\frac{value(t_2) \qquad subst(x, t_2, t_1, t)}{eval((\lambda x : \tau.t_1)\ t_2, t)}\ \text{E-App-Subst}$$

$$\frac{eval(t_1, t_1')}{eval(t_1 + t_2, t_1' + t_2)}\ \text{E-Plus-Step1} \qquad \frac{value(t_1) \qquad eval(t_2, t_2')}{eval(t_1 + t_2, t_1 + t_2')}\ \text{E-Plus-Step2}$$

$$\frac{i_1 + i_2 = i}{eval(int(i_1) + int(i_2), int(i))}\ \text{E-Plus-Add}$$

$\boxed{value(tm^*)}$

$$\frac{}{value(\lambda x : \tau.t)}\ \text{V-Abs} \qquad \frac{}{int(i)}\ \text{V-Int}$$

Figure 7.2: Syntax for our lambda calculus host language, its evaluation relation and rules defining it, and its predicate and rules defining value forms

Let extension:

$$\frac{\Gamma \vdash t_1 : \tau}{proj_{tm}(\Gamma, let(x, t_1, t_2), (\lambda x : \tau.t_2)\ t_1)}\ \text{P-Let}$$

Pair extension:

$$\frac{\Gamma \vdash t_1 : \tau}{proj_{tm}(\Gamma, pair(t_1, t_2), (\lambda a : \tau.\lambda b : \tau.\lambda s : \tau \to \tau \to \tau.s\ a\ b)\ t_1\ t_2)}\ \text{P-Pair}$$

$$\frac{\Gamma \vdash t : pairTy(\tau)}{proj_{tm}(\Gamma, fst(t), t\ (\lambda a : \tau.\lambda b : \tau.a))}\ \text{P-Fst} \qquad \frac{\Gamma \vdash t : pairTy(\tau)}{proj_{tm}(\Gamma, snd(t), t\ (\lambda a : \tau.\lambda b : \tau.b))}\ \text{P-Snd}$$

$$\frac{}{proj_{ty}(pairTy(\tau), (\tau \to \tau \to \tau) \to \tau)}\ \text{P-PairTy}$$

Figure 7.3: Projection rules for extensions to the simply-typed lambda calculus

in detail, such as that typing, substitution, and evaluation are unique; terms identified as values cannot take an evaluation step; type preservation for evaluation and substitution; and totality of substitution. All of these proofs proceed as one would expect in a non-extensible setting, with the only difference being the distribution of the proof cases across the three modules. We will focus on one property introduced by the host language and the lemmas necessary for proving it that cannot be proven exactly as in a non-extensible setting. The property is progress, that terms well-typed with an empty typing context are either values or can take an evaluation step:

$$\forall t, \tau. \cdot \vdash t : \tau \supset (\exists t'. eval(t, t')) \vee value(t) \tag{7.1}$$

The lemmas are *canonical form lemmas*, that values with particular types have particular forms. One such lemma our host language can introduce and prove is

$$\forall t. \cdot \vdash t : intTy \supset value(t) \supset \exists i. t = int(i) \tag{7.2}$$

stating that the form of a term identified as a value and that has an integer type is that of a numeric constant using the constructor *int*.

The proof of progress typically proceeds by induction and case analysis on the typing derivation, in each case either noting the term is a value form or applying the induction hypothesis to its sub-terms and checking whether they are values or can take evaluation steps. If one of the sub-terms can take a step of evaluation, the appropriate rule for evaluating the whole term by that sub-term stepping can be used. If not, all sub-terms are values, and we use the appropriate canonical form lemma to show the values are the right shape to use another rule. For example, in proving progress for an addition $t_1 + t_2$, we use the induction hypothesis to show that either t_1 or t_2 steps, or they are both values. If t_1 steps to t_1', we can use the E-Plus-Step1 rule from Figure 7.2 to find $t_1 + t_2$ steps to $t_1' + t_2$, and similarly use the E-Plus-Step2 rule if t_2 steps. If both are values, our canonical form lemma for integers, shown above, tells us both t_1 and t_2 must be numeric constants because they both

Let:

$$\frac{eval(t_1, t_1')}{eval(let(x, t_1, t_2), let(x, t_1', t_2))}\ \text{E-Let-Step} \qquad \frac{value(t_1) \qquad subst(x, t_1, t_2, t)}{eval(let(x, t_1, t_2), t)}\ \text{E-Let-Subst}$$

Pair:

$$\frac{eval(t_1, t_1')}{eval(pair(t_1, t_2), pair(t_1', t_2))}\ \text{E-Pair-Step1} \qquad \frac{eval(t_2, t_2')}{eval(pair(t_1, t_2), pair(t_1, t_2'))}\ \text{E-Pair-Step2}$$

$$\frac{eval(t, t')}{eval(fst(t), fst(t'))}\ \text{E-Fst-Step} \qquad \frac{value(pair(t_1, t_2))}{eval(fst(pair(t_1, t_2)), t_1)}\ \text{E-Fst-Pair}$$

$$\frac{eval(t, t')}{eval(snd(t), snd(t'))}\ \text{E-Snd-Step} \qquad \frac{value(pair(t_1, t_2))}{eval(snd(pair(t_1, t_2)), t_2)}\ \text{E-Snd-Pair}$$

Figure 7.4: Evaluation rules from the let and pair extensions

have type *intTy*. We can then use the E-Plus-Add rule from Figure 7.2 to evaluate the addition.

The proof of progress for the let extension proceeds similarly to that of the host language. For $let(x, t_1, t_2)$, either t_1 steps to t_1', giving us $let(x, t_1', t_2)$ by the E-Let-Step rule in Figure 7.4, or t_1 is a value, in which case we use the E-Let-Subst rule, substituting t_1 into t_2 for x. Note we do not care what value form t_1 has; it could be a numeric constant, an abstraction, or a pair as introduced by the other extension. That it does not care about the form is a key reason why the let extension does not encounter any problems in its proof.

The pair extension, in contrast, does have rules that depend on a particular value form, specifically the new value form it introduces, and thus it does encounter problems in proving progress. The evaluation rules for the pair extension are also shown in Figure 7.4. The evaluation rules for $fst(t)$ either step t to t' (E-Fst-Step) or, if t is a value, expect it to have the form $pair(t_1, t_2)$ (E-Fst-Pair); the rules for *snd* are similar. The proof of progress for $fst(t)$ uses the induction hypothesis to show that either t steps, in which case we can use the E-Fst-Step rule, or it is a value. If t is a value, we need it to be constructed by the *pair* constructor in order to apply the E-Fst-Pair rule, just as we needed the sub-terms to be numbers to apply the E-Plus-Add rule for the host language's *add* constructor. To

show t must be *pair*-shaped, we need a canonical form lemma for the pair type:

$$\forall t, \tau. \cdot \vdash t : \mathit{pairTy}(\tau) \supset \mathit{value}(t) \supset \exists t_1, t_2. t = \mathit{pair}(t_1, t_2) \tag{7.3}$$

Note that such a property must be introduced by the pair extension, since it mentions the *pair* and *pairTy* constructors introduced by it; the host language does not know these constructors, and thus cannot introduce this property.

Unfortunately, whether we use the value predicate or the typing relation as our key relation for this canonical form lemma, we must prove a case for a generic constructor κ due to the key relation being an imported one, and we cannot. In the generic case, we assume κ has type $\mathit{pairTy}(\tau)$ and that it is a value. Because the generic constructor, standing in for a term built by a new constructor from another extension, cannot have *pair* as its top-level symbol, the proof requires showing this combination of assumptions is impossible, but we have no way to show this. The projection of κ does not need to have the same type, nor does it need to be a value, so we cannot prove the lemma in the generic case, leaving the pair extension's proof of progress stuck.

Without extensions being able to introduce new canonical form lemmas, it appears our language library cannot support both extensions introducing new value forms and the host language introducing the progress property. Fortunately, the host language designer can recognize the problems that will arise for extension modules introducing new canonical form lemmas and introduce a solution before any extensions are written. Rather than having each module introduce a new canonical form lemma specific to each of its new types, the host language declares an extensible canonical form relation *canon*, shown in Figure 7.5, relating a type to terms of the form expected for its values. The unified canonical form lemma introduced by the host language then states a value-identified, typable term has the *canon* relation hold for its type and itself:

$$\forall t, \tau. \cdot \vdash t : \tau \supset \mathit{value}(t) \supset \mathit{canon}(\tau, t)$$

$\boxed{canon(ty^*, tm) \in \mathscr{R}^{host}}$

$\mathbb{R}^{host}$:

$$\frac{}{canon(intTy, int(i))}\ \text{C-Int} \qquad \frac{}{canon(\tau_1 \to \tau_2, \lambda x : \tau_1.t)}\ \text{C-Arrow}$$

$\mathbb{R}^{pair}$:

$$\frac{}{canon(pairTy(\tau), pair(t_1, t_2))}\ \text{C-Pair}$$

Figure 7.5: A canonical form relation to define one overarching canonical form lemma

We can see how this corresponds to the forms of Properties 7.2 and 7.3 for integers and pairs. When the type τ is an integer type, the term must have an integer form, as specified by the C-Int rule, and when the type is a pair type, the term must have a pair form, as specified by the C-Pair rule. Because it is introduced by the host language, which also introduces its key relation and the primary component category thereof, we do not need to prove a generic case as we would need to do for the pair-specific version introduced by the pair extension, avoiding the difficulties we saw there.

Consider proving progress for addition again. If both sub-terms of $t_1 + t_2$ are values, we can use the unified canonical form lemma and the fact that both must have type *intTy* to get $canon(intTy, t_1)$ and $canon(intTy, t_2)$. Because the type is the primary component of *canon*, we can analyze these to find both t_1 and t_2 are numeric constants by the C-Int rule, and thus we can use the E-Plus-Add rule to evaluate the original addition. Similarly, for $fst(t)$ from the pair extension, if t is a value, it must have type $pairTy(\tau)$, and our unified canonical form lemma gives us $canon(pairTy(\tau), t)$. Case analysis on this shows t has a *pair* shape by the C-Pair rule, and we can apply the E-Fst-Pair rule to evaluate $fst(t)$. We choose the type as the primary component of the *canon* relation both because this is how we need to use it, to find the form of terms when the type is known, and because the purpose of the relation is to tell us something *about* the type, namely what forms values for it might take.

By the host language designer recognizing the difficulties extensions will face with canonical form lemmas and introducing a relation and property to alleviate them, we are able to have both the progress property and freedom for extensions to introduce new value forms. This approach could be adapted to other situations where the host language can anticipate a certain form of lemma, where different constructs need the same type of information but the details are specific to each construct, is needed by extensions to support the host language's properties, allowing it to unify all the lemmas into one that it introduces itself.

We also see in this example how extensibility can affect both language specifications and property statements. In a non-extensible setting, there would not be a reason to have the *canon* relation, as all canonical form lemmas could be written and proven directly. It is only in the extensible setting that we need to encapsulate them so.

7.3 Lambda Calculus with Typing Extension

Next we consider a language library with a different version of the lambda calculus, one where typing is introduced by an extension, and examine how this change of module structure affects what we may prove about typing. The new host language is similar to that from the previous section, just without typing.[2] The new language library includes a pair extension, similar to the one from the language library of the previous section, allowing programmers using a language to use pair syntax. As mentioned, it also includes an extension introducing typing to the language. Including typing as an extension instead of as part of the host language allows users to choose whether they want static typing to give them some safety or would prefer the freedom to use any term the syntax allows, not just typable ones. This change allows us to consider the effects of introducing a relation in an extension, instead of in the host language that introduces its primary component category, on its definition and on what may be proven about it. Specifically, this change means the typing relation will be defined in some cases by a default rule, and properties about it will have generic cases to be proven.

[2]Sterling and Extensibella code available online at `https://mmel.cs.umn.edu/extensibella/examples/lambda_calculus/description.html`

$$
\begin{array}{rcl}
tm & ::= & \lambda x.tm \\
& | & tm\ tm \\
& | & x \\
& | & tm + tm \\
& | & int(i)
\end{array}
$$

$\boxed{eval(tm^*, tm)}$

$$\frac{}{eval(\lambda x.t, \lambda x.t)}\ \text{E-Abs} \qquad \frac{}{eval(int(i), int(i))}\ \text{E-Int}$$

$$\frac{eval(t_1, \lambda x.t_1') \qquad eval(t_2, t_2') \qquad subst(x, t_2, t_1', t') \qquad eval(t', t)}{eval(t_1\ t_2, t)}\ \text{E-App}$$

$$\frac{eval(t_1, int(i_1)) \qquad eval(t_2, int(i_2)) \qquad i_1 + i2 = i}{eval(t_1 + t_2, int(i))}\ \text{E-Plus}$$

Figure 7.6: Syntax and evaluation rules for the untyped lambda calculus

As in the previous section, the host language defines syntax for terms and a substitution relation, with the syntax shown in Figure 7.6. Note that we no longer have syntax for types, nor does the abstraction have a type annotation for its argument, as the host language does not introduce typing. The host language introduces substitution for variables (relation *subst*), with the rules being as expected, and big-step evaluation, with rules shown in Figure 7.6. It introduces a projection relation for the *tm* syntax category that takes no extra arguments. Using this projection relation, it defines a proxy rule for the evaluation relation:

$$\frac{proj_{tm}(t, t_0) \qquad eval(t_0, t'')}{eval(t, t')}\ \text{E-Q}$$

This allows extensions introducing properties with evaluation as the key relation to assume, in the generic case, that the term being evaluated projects and its projection also evaluates.

The host language introduces properties that substitution is total and evaluation and substitution are unique, with these proofs proceeding as in non-extensible settings. It also introduces projection constraints about evaluation and substitution. The projection constraints this host language introduces are that substitution and evaluation are exactly

the same for a construct and its projection, stated as

$$\forall x, r, t, t', t_0.subst(x, r, t, t') \supset proj_{tm}(t, t_0) \supset subst(x, r, t_0, t')$$

for substitution and

$$\forall t, v, t_0.eval(t, v) \supset proj_{tm}(t, t_0) \supset eval(t_0, v)$$

for evaluation, that a term t's projection t_0 evaluates to the same value and substituting for a variable in t and t_0 produces the same substituted term. Finally, the host language introduces the $addP(eval)$ property without any extra premises (*i.e.*, in the terminology of Section 4.4.3, F_{eval} is empty).

We introduce a pair extension as in the previous section, with constructors for building pairs and selecting elements from them (*i.e.*, the same constructors as in the previous section other than the *pairTy* constructor). The rules from this extension are shown in Figure 7.7. The projection for each constructor is the same as in the previous section, but without types for the abstractions. Looking at rules S-Pair, S-Fst, and S-Snd that define substitution for the pair extension's constructs, the definitions appear a bit peculiar. Rather than substitution in a pair $pair(t_1, t_2)$ resulting in a pair $pair(s_1, s_2)$ where substitution in t_1 gives s_1 and substitution in t_2 gives s_2, we instead find the substitution results in applications of an abstraction. This is because the host language's projection constraint for substitution requires the pair's substitution give exactly the same term as the substitution in its projection. We can see the result has the same structure as the pair's projection, but with the substituted sub-terms replacing the original ones, and that the same is true for the rules for substitution in *fst* and *snd*. The evaluation relation is similarly constrained, and the pair extension's rules defining it take a more direct approach to showing the correlation, defining the evaluation of pair constructs directly as the evaluation of their projections. Thus we see the host language's projection constraints are very strict, eliminating the freedom of extensions to introduce their own value constructors and thus requiring odd definitions.

$\boxed{proj_{tm}(tm^*, tm)}$

$$\frac{}{proj_{tm}(pair(t_1, t_2), (\lambda a.\lambda b.\lambda s.s\ a\ b)\ t_1\ t_2)}\ \text{P-Pair}$$

$$\frac{}{proj_{tm}(fst(t), t\ (\lambda a.\lambda b.a))}\ \text{P-Fst} \qquad \frac{}{proj_{tm}(snd(t), t\ (\lambda a.\lambda b.b))}\ \text{P-Snd}$$

$\boxed{subst(x, tm, tm^*, tm)}$

$$\frac{subst(x, r, t_1, s_1) \qquad subst(x, r, t_2, s_2)}{subst(pair(t_1, t_2), (\lambda a.\lambda b.\lambda s.s\ a\ b)\ s_1\ s_2)}\ \text{S-Pair}$$

$$\frac{subst(x, r, t, s)}{subst(x, r, fst(t), s\ (\lambda a.\lambda b.a))}\ \text{S-Fst} \qquad \frac{subst(x, r, t, s)}{subst(x, r, snd(t), s\ (\lambda a.\lambda b.b))}\ \text{S-Snd}$$

$\boxed{eval(tm^*, tm)}$

$$\frac{eval((\lambda a.\lambda b.\lambda s.s\ a\ b)\ t_1\ t_2, t)}{eval(pair(t_1, t_2), t)}\ \text{E-Pair}$$

$$\frac{eval(t\ (\lambda a.\lambda b.a), t')}{eval(fst(t), t')}\ \text{E-Fst} \qquad \frac{eval(t\ (\lambda a.\lambda b.b), t')}{eval(snd(t), t')}\ \text{E-Snd}$$

Figure 7.7: Rules introduced by the pair extension

The benefit of these strict constraints is that they allow other extensions to understand evaluation very well, as we shall see in considering another extension, this one introducing typing to the language. It gives a new syntax category for types and constructors *arrowTy* and *intTy* for it, as well as a syntax constructor for abstractions with their variables annotated with types (*i.e.*, abstractions as we see in the simply-typed lambda calculus) projecting to the host language's untyped abstractions, and defines evaluation and substitution for it. Most importantly, it introduces a typing relation, giving rules for the host language's constructs and its own typed abstraction, which are as expected, with the default rule

$$\frac{proj_{tm}(t, t') \qquad \Gamma \vdash t' : \tau}{\Gamma \vdash t : \tau} \text{ T-DEFAULT}$$

This copies the type for an unknown construct, such as the pair extension's *pair* constructor, from its projection.

This extension also introduces several properties about typing, culminating in a proof of type preservation with evaluation as its key relation:

$$\forall t, \tau, t'.eval(t, t') \supset \cdot \vdash t : \tau \supset \cdot \vdash t' : \tau$$

The cases for known rules proceed as in non-extensible settings, noting each sub-term evaluates to a value of the correct type, and thus the whole term evaluates to the correct type. In the generic case, which is for the proxy rule E-Q for evaluation instantiated for the generic constructor κ, we know that κ evaluates to t', κ projects to t_0 and that t_0 evaluates to t''. We can analyze the typing derivation, which must be derived by the T-DEFAULT rule instantiated for κ, to find t_0 has the same type, then use the induction hypothesis to find the projection's value t'' has the same type as well. Since both the original term and its projection are required by the host language's projection constraint to evaluate to the same value ($t' = t''$), this completes the proof in the generic case.

In the relationship between the projection constraints, the definitions given by the pair extension, and the property proven by the typing extension, we see the trade-off between

extension freedom and the ability to prove desired properties. By the host language restricting how extensions define their semantics, the pair extension's rules end up a bit strange, but the typing extension can prove type preservation. If the host language gave extensions more freedom by having less-restrictive projection constraints, the pair extension's evaluation rules could match those from the previous section, but the typing extension would find it difficult, if not impossible, to prove its property.

Despite the apparent strictness of the host language, in some sense extensions have more freedom than the typing extension would like, as we cannot prove progress with these constraints. In the generic case we would be able to show the projection evaluates, but without a projection constraint saying a term evaluates whenever its projection evaluates, we cannot lift the evaluation back to the original term. However, limiting freedom more, while it happens it would not affect the pair extension, would mean some extensions that could be written in the current language would be disallowed.

Considering this example relative to the previous one, we also see how which modules introduce relations and properties affects their definitions and provability. With typing introduced by the host language in the previous section, which also introduced its key relation and primary component category, proving progress was not a problem because all modules knew the property and thus it could always be treated non-generically. Having the host language introducing typing also allows the pair extension to define its own type and typing, giving a better definition of typing for pairs. In this language, where typing is introduced by an extension, we need to treat type preservation generically, which requires strict constraints, and would need to treat progress generically with even stricter constraints to be able to prove it. Moving typing to an extension also means the pair extension cannot introduce pair-specific typing, resulting in a subjectively worse definition of typing overall (*e.g.*, $fst(\lambda x.x)$ would be typable because the projection of *fst* expects a function). Thus we see that while some analyses do not significantly suffer from being introduced by extension modules, such as the relations from the security and optimization extensions from Chapter 2, others have worse definitions, and significant restrictions on what properties can be proven about them.

Whether a relation's definition suffers from being introduced by a different module than introduces its primary component category appears to be tied to the exact nature of the relation. Typing, which relates specific constructor forms for expressions and types, works best when each expression form can have rules written explicitly for it. Information flow security for expressions, which depends only on the variables in an expression and the levels they are assigned, can be defined just as well generically as directly. Optimization for expressions falls somewhere in between. It would be useful to have rules written explicitly for each expression form. However, our default rule that optimizes an expression to itself also gives a workable definition, one that is perhaps better than the definition of typing through a default rule in this language, but worse than the ideal definition.

7.4 Variations on an Imperative Language

Our final example investigates the trade-offs in the strictness of projection constraints and the freedom of extensions in defining their semantics. It also looks at how the constructs included in a host language affect the constructs and semantics extensions can introduce, particularly in light of differing projection constraints. In particular, we find that both the host language's choices of metatheoretic properties, especially projection constraints, and its choices in defining its own syntax and semantics affect what extensions may introduce and how extensions must write their definitions. We also find that the strictness of projection constraints can affect not only what properties extensions can prove, but also how the proofs of those properties are written.

Our investigation uses three variations of a language and its metatheory that differ in minor ways in the constructs they introduce, and differ in major ways in the projection constraints they introduce. Using three language libraries with similar host languages and adding similar extensions to them lets us directly compare how the differences affect what constructs extensions can introduce, how they can define the semantics of those constructs, and the properties they can prove.

The language underlying all three variations is an imperative language, similar to the

one from Chapter 2, but more complex. It has expression forms for numbers and arithmetic, booleans, strings, records, and function calls and type forms for integer, boolean, string, and record types. It also has an expression form *error* representing situations where computation fails, the idea being it is like throwing an exception in a language like Java and halts execution. Our underlying language includes the statement forms from the language in Chapter 2, *skip*, *decl*, *assign*, *seq*, *ifte*, and *while*. We additionally include three new constructs. The *print* construct (*print*(e)) represents printing the value of an expression, and the *recUpdate* (*recUpdate*(n, ns, e) where *ns* is a sequence of field names) represents updating a field in an existing record (*e.g.*, `r.f1.f2 = e`, using concrete syntax like C's struct updates). Finally, the *scopeStmt* construct (*scopeStmt*(s)) represents opening a new scope for names. The scope of a variable name is the part of a program in which it is valid. In our language, as in languages like C and Java, we generalize this to a scope being an explicitly-created portion of the program where names declared inside it may be used inside it after their declarations but are then not available outside.

In addition to expressions, statements, and types, our language includes syntax categories for function declarations *fun* and programs *prog*, each with one constructor. A function declaration has a name for the function, parameters to the function and their types, a return type, and a statement body. A program is a list of function declarations, with one chosen as the main function, as in C or Java.

We have six extensions, summarized in Table 7.1. We split these into two groups. Syntactic extensions are those focused on adding new syntax, not new properties or relations; new properties and relations added in them are there only to support the new syntax and extend the imported semantics to it. Semantic extensions are focused on adding new relations and properties, not new syntax, although they may do so to support making use of their new relations and properties. In the remainder of this section, we first look at the differences between the language libraries, then look at how the differences affect the implementations of each of these modules as extensions in the libraries.

Syntactic Extensions	
Ascription Extension	Adds an expression form for ascribing types to arbitrary extensions
Assertion Extension	Adds a statement form for asserting conditions are true and abruptly terminating if they are not
Conditional Expression Extension	Adds an expression form for conditionally evaluating one of two expressions
List Extension	Adds list expression forms like *cons* and *tail*, statement forms *listUpdate* for updating a list in place and *listForeach* for looping over all elements of a list, and a list type
Semantic Extensions	
Security Extension	Adds a security analysis to check programs do not leak sensitive information
Translation Extension	Adds a translation relation to translate programs into the host language

Table 7.1: Extension modules added to our language variations

7.4.1 Language Library Versions and Projection Constraints

We have three versions of our language and metatheory. They differ in the particulars of how their host languages define the language semantics, with one also introducing another expression form to make its language richer. They also differ in how strict their projection constraints for evaluation are. We look at each in turn.

Loose Evaluation in V_L

Our first version of the language library has loose projection constraints for expression evaluation, so we use V_L to refer to it and its metatheory. Some of its relevant relations and a few rules are shown in Figure 7.8.[3] These relations make use of some non-extensible syntax categories that will be shared by all three versions of the underlying language. These categories are:

- *fECtx*: A context of known functions that carries information necessary for function call expressions to lookup the function being called and evaluate its body.

[3]The full Sterling and Extensibella development can be found at `https://mmel.cs.umn.edu/extensibella/examples/looseEval/description.html`.

- *output*: A sequence of printed values that has constructors *nilOutput* with no arguments and $consOutput(e, output)$.

- *eCtx*: Scoped evaluation contexts that map variable names to values. The constructors for this category are *nilScopes* for completely empty contexts and $addScope(\gamma, eCtx)$ where γ is a set of bindings as in the language from Chapter 2; recall γ has constructors *nilval* and $consval(n, e, \gamma)$.

These categories help us define evaluation with function calls, printed output, and explicit scopes for variables.

Our language includes a number of relations. The ones we will discuss are listed in Figure 7.8. Note that for relations introduced by different language libraries that have different definitions in the different libraries, we will subscript uses of them to specify to which library they belong. One relation introduced by V_L, $vars_L$, maps expressions to the variables occurring in them. The expression evaluation relation $evalExpr_L$ relates a function context, evaluation context, and expression to its produced value and printed output. The statement evaluation relation $evalStmt_L$ relates a function context, evaluation context, and statement to an updated evaluation context and printed output. The program evaluation relation $evalProg_L$ relates a list of arguments and a program to its printed output, being defined as evaluating its main function with the arguments given. The projection relation for statements includes another argument besides the projection statement and its projection, this argument being a set of variables. This is the set of names currently bound in the program at the point the statement occurs, which we can get from an evaluation context using the $namesOf(eCtx^*, 2^n)$ relation. Using the set of known names, we can find fresh ones with the $freshName((2^n)^*, n)$ relation, which, assuming an ordering on variable names, finds the first name not in the given set, letting statement projections use new temporary variables. Note these last two relations are defined the same across all three libraries, so we do not subscript them.

The two interesting aspects of the relations here compared to those from Chapter 2 are the explicit scoping in evaluation contexts and the printed output, both seen in rules

Relations include

$vars_L(e^*, 2^n)$, $evalExpr_L(fECtx, eCtx, e^*, e, output)$,
$evalStmt_L(fECtx, eCtx, s^*, eCtx, output)$, $evalProg_L(\overline{e}, prog^*, output)$,
$namesOf(eCtx^*, 2^n)$, $freshName((2^n)^*, n)$, $concat(output^*, output, output)$

Projection relations include

$proj^e_L(e^*, e)$, $proj^s_L(2^n, s^*, s)$, $proj^{prog}_L(prog^*, prog)$

$\boxed{evalExpr_L(fECtx, eCtx, e^*, e, output)}$

$$\frac{}{evalExpr_L(fECtx, eCtx, intlit(i), intlit(i), nilOutput)}\ \text{E}_L\text{-Intlit}$$

$$\frac{\begin{array}{c}evalExpr_L(fECtx, eCtx, e_1, intlit(i_2), o_1)\\ evalExpr_L(fECtx, eCtx, e_2, intlit(i_1), o_2)\\ plus(i_1, i_2, i) \qquad concat(o_1, o_2, o)\end{array}}{evalExpr_L(fECtx, eCtx, add(e_1, e_2), intlit(i), o)}\ \text{E}_L\text{-Add}$$

$\boxed{evalStmt_L(fECtx, eCtx, s^*, eCtx, output)}$

$$\frac{\begin{array}{c}evalExpr_L(fECtx, eCtx, e, true, o_1)\\ evalStmt_L(fECtx, addScope(nilval, eCtx), s_t, addScope(\gamma, eCtx'), o_2)\\ concat(o_1, o_2, o)\end{array}}{evalStmt_L(fECtx, eCtx, ifte(e, s_t, s_f), eCtx', o)}\ \text{X}_L\text{-If-True}$$

$$\frac{evalStmt_L(fECtx, addScope(nilval, eCtx), s, addScope(\gamma, eCtx'), o)}{evalStmt_L(fECtx, eCtx, scopeStmt(s), eCtx', o)}\ \text{X}_L\text{-ScopeStmt}$$

$$\frac{\begin{array}{c}evalStmt_L(fECtx, eCtx, s_1, eCtx_1, o_1)\\ evalStmt_L(fECtx, eCtx_1, s_2, eCtx', o_2) \qquad concat(o_1, o_2, o)\end{array}}{evalStmt_L(fECtx, eCtx, seq(s_1, s_2), eCtx', o)}\ \text{X}_L\text{-Seq}$$

$$\frac{\begin{array}{c}evalExpr_L(fECtx, eCtx, e, intlit(i), o')\\ concat(o', consOutput(intlit(i), nilOutput), o)\end{array}}{evalStmt_L(fECtx, eCtx, print(e), eCtx, o)}\ \text{X}_L\text{-Print-Int}$$

Figure 7.8: Selected relations and rules in V_L's host language

in Figure 7.8. We see the explicit scoping in X_L-If-True and X_L-ScopeStmt. In both rules we create a new, empty scope for evaluating a sub-statement, and remove the scope after its evaluation, eliminating the new bindings that occurred in the scope but keeping any updates to bindings from the rest of the program. We see specifications of printed output in all the rules. Note that, while expressions cannot directly print themselves, as printed output only comes from *print* statements, they may indirectly print through function calls. Evaluating numeric constants cannot print anything, so the output in the E_L-Intlit rule is empty. In the E_L-Add rule, both sub-expressions may print, so we use a relation $concat(output^*, output, output)$ that concatenates the two lists into one. We similarly concatenate output in X_L-Seq and X_L-If-True. In X_L-Print-Int, the output is whatever the expression's evaluation printed plus the integer value being printed.

The host language for V_L introduces a number of properties; the ones that will be important in our discussion are primarily the projection constraints. The looseness of projection constraints here refers primarily to expression evaluation. The host language introduces a constraint that an expression's projection evaluates if it does, and that they have the same printed output:

$$\forall e, e', fECtx, eCtx, v, o. proj^e_L(e, e') \supset$$
$$evalExpr_L(fECtx, eCtx, e, v, o) \supset \exists v'. evalExpr_L(fECtx, eCtx, e', v', o)$$

This does not place any requirements on values produced by evaluating each; as we shall see in our discussions of extensions, they can be totally unrelated. However, by requiring the same printed output from both, it places an implicit constraint on the order in which sub-expressions are to be evaluated in both. Because we concatenate printed output in rules based on which sub-expression is evaluated "first", conceptually, an expression's projection must evaluate the sub-expressions in the same order. Note this projection constraint does not apply to printing values and their projections themselves, such as evaluating $print(e)$ and $print(e')$ where e projects to e', only to the printed output of evaluating each of e and e'.

The host language for V_L imposes a stricter constraint on statement evaluation, that a statement's projection evaluates with the same printed output and the same updated evaluation context:

$$\forall s, s', ns, \mathit{fECtx}, \mathit{eCtx}, \mathit{eCtx}', o.\mathit{proj}^s_L(ns, s, s') \supset$$
$$\mathit{evalStmt}_L(\mathit{fECtx}, \mathit{eCtx}, s, \mathit{eCtx}', o) \supset \mathit{evalStmt}_L(\mathit{fECtx}, \mathit{eCtx}, s', \mathit{eCtx}', o) \quad (7.4)$$

This initially appears perhaps too strict to be compatible with our projection constraint for expressions. However, it only applies when projecting *statements*, which does not mean projecting the expressions within them, so the two constraints are separate. Because the host language's statement forms can implement any sort of control flow an extension might want, expecting the same behavior from extension-introduced statements and their projections does not end up too strict.

As an example of how the constraints for expressions and statements are separate, consider the $\mathit{splitlist}(n_{hd}, n_{tl}, e)$ construct from the list extension from Chapter 2 representing simultaneous assignment of the head of e's result to n_{hd} and the tail of e's result to n_{tl}. This projects to a sequence of assignments using the list extension's *head* and *tail* constructs:

$$\mathit{seq}(\mathit{seq}(\mathit{assign}(n_{hd}, e), \mathit{assign}(n_{tl}, \mathit{tail}(\mathit{var}(n_{hd})))), \mathit{assign}(n_{hd}, \mathit{head}(\mathit{var}(n_{hd}))))$$

The projection of the *splitlist* still uses the same expression e, not its projection. Under the projection, this e still evaluates to the same value. This value must have the form $\mathit{cons}(e_1, e_2)$ in order for the original *splitlist* to evaluate, and then the *head* and *tail* in the projection evaluate as well. The *splitlist* construct then satisfies this projection constraint, even though the list expressions on which it operates project to unrelated things, as our projection constraint for expressions allows. Thus, because statement projections do not also need to project the expressions within them, the strictness of the projection constraints for expressions and statements are independent.

The host language also introduces projection constraints that two projections of a term

are the same (*i.e.*, projection is unique). For projecting statements, this is qualified by having the same set of variables used:

$$\forall ns, s, s_1, s_2.proj^s_L(ns, s, s_1) \supset proj^s_L(ns, s, s_2) \supset s_1 = s_2$$

Note that because the *freshName* relation finds the *first* name not in the given set, assuming a fixed ordering of variable names, it is also unique, always relating a set of variables to the same fresh one. Thus we can have both uniqueness of statement projection and fresh variables used as temporaries in a projection.

Another projection constraint the host language introduces is that the variables in an expression's projection are the same as those in the original expression:

$$\forall e, e', v, v'.proj^e_L(e, e') \supset vars_L(e, v) \supset vars_L(e', v') \supset v = v' \tag{7.5}$$

This, along with the requirement for evaluation having the same printed output, means extensions cannot drop sub-expressions in projecting, as that might change the set of variables or printed output.

The host language for V_L also introduces a property that expression evaluation is determined entirely by the values assigned to its variables in the evaluation context:

$$\begin{aligned}\forall e, ns, fECtx, eCtx_1, eCtx_2, v, o.evalExpr_L(fECtx, eCtx_1, e, v, o) \supset vars_L(e, ns) \supset \\ (\forall x, xv.x \in ns \supset lkpVal(eCtx_1, x, xv) \supset lkpVal(eCtx_2, x, xv)) \supset \\ evalExpr_L(fECtx, eCtx_2, e, v, o)\end{aligned} \tag{7.6}$$

This is stated as if an expression evaluates under $eCtx_1$ and every variable x in the expression's set of variables has the same value in $eCtx_2$ as in $eCtx_1$, then the expression evaluates under $eCtx_2$ with the same value and printed output.

Finally, V_L introduces the $addP(evalExpr_L)$ and $addP(evalStmt_L)$ properties. The proxy

rule for the $evalExpr_L$ relation is

$$\frac{proj^e_L(e, e') \qquad evalExpr_L(fECtx, eCtx, e', v', o)}{evalExpr_L(fECtx, eCtx, e, v, o)}\ \text{E}_L\text{-Q}$$

This requires an expression with an extension-introduced constructor as its top-level symbol project whenever it evaluates, and that the projection also evaluate. The proxy rule for statement evaluation is similar:

$$\frac{namesOf(eCtx, ns) \qquad proj^s_L(ns, s, s') \qquad evalStmt_L(fECtx, eCtx, s', eCtx', o)}{evalStmt_L(fECtx, eCtx, s, eCtx', o)}\ \text{X}_L\text{-Q}$$

This also requires evaluating statements with extension-introduced top-level symbols to project and those projections to evaluate, but is also qualified by taking the set of known names from the evaluation context to use to create the projection.

Evaluation and Projection in V_P

Our next version of the language library has a stricter projection constraint for expression evaluation than V_L, one that will require a relationship between values based on projection, so we refer to it as V_P.[4] It introduces the same non-extensible syntax categories and relations over them as V_L introduces. Figure 7.9 gives some of the other relations in the language and some rules defining them. The evaluation relations have the same types as in V_L, but we see some of the rules are defined differently. While the E_P-Intlit rule is the same as the E_L-Intlit rule, the E_P-Add rule uses a *matchInt* relation to get integers from its sub-expressions' values rather than assuming the sub-expressions evaluate to values constructed by *intlit* directly. To understand the purpose of this matching, we need to understand the projection constraint for expression evaluation V_P introduces, as it forces this choice.

Unlike V_L, which does not require a relationship between the value of an expression

[4]The full Sterling and Extensibella development can be found at `https://mmel.cs.umn.edu/extensibella/examples/matchEval/description.html`.

Relations include

$vars_P(e^*, 2^n)$, $evalExpr_P(fECtx, eCtx, e^*, e, output)$,
$evalStmt_P(fECtx, eCtx, s^*, eCtx, output)$, $evalProg_P(\overline{e}, prog^*, output)$,
$matchInt(e^*, i)$, $matchTrue(e^*)$, $matchFalse(e^*)$, $matchRec(e^*, fs)$

Projection relations include

$proj^e_P(e^*, e)$, $proj^s_P(2^n, s^*, s)$, $proj^{prog}_P(prog^*, prog)$

$\boxed{evalExpr_P(fECtx, eCtx, e^*, e, output)}$

$$\frac{}{evalExpr_P(fECtx, eCtx, intlit(i), intlit(i), nilOutput)}\ \text{E}_P\text{-Intlit}$$

$$\frac{\begin{array}{c}evalExpr_P(fECtx, eCtx, e_1, v_1, o_1) \qquad evalExpr_P(fECtx, eCtx, e_2, v_2, o_2) \\ matchInt(v_1, i_1) \qquad matchInt(v_2, i_2) \qquad plus(i_1, i_2, i) \qquad concat(o_1, o_2, o)\end{array}}{evalExpr_P(fECtx, eCtx, add(e_1, e_2), intlit(i), o)}\ \text{E}_P\text{-Add}$$

$\boxed{matchInt(e^*, i)}$

$$\frac{}{matchInt(intlit(i), i)}\ \text{MI-Intlit}$$

$\boxed{matchTrue(e^*)}$

$$\frac{}{matchTrue(true)}\ \text{MT-True}$$

$\boxed{matchRec(e^*, fs)}$

$$\frac{}{matchRec(rec(fs), fs)}\ \text{MR-Rec}$$

Figure 7.9: Selected relations and rules in V_P's host language

$\boxed{projedVal(e^*, e)}$

$$\frac{}{projedVal(e, e)}\ \text{PV-Zero} \qquad \frac{projedVal(e_1, e_2) \qquad projedVal(e_2, e_3)}{projedVal(e_1, e_3)}\ \text{PV-Trans}$$

$$\frac{proj^e_P(e_1, e_2)}{projedVal(e_1, e_2)}\ \text{PV-Proj} \qquad \frac{projedFields(f_1, f_2)}{projedVal(rec(f_1), rec(f_2))}\ \text{PV-Rec}$$

$\boxed{projedFields(fs^*, fs)}$

$$\frac{}{projedFields(nilFs, nilFs)}\ \text{PF-Nil}$$

$$\frac{projedVal(e_1, e_2) \qquad projedFields(fs_1, fs_2)}{projedFields(consFs(n, e_1, fs_1), consFs(n, e_2, fs_2))}\ \text{PF-Cons}$$

Figure 7.10: Rules for *projedVal* relation for value expressions and *projedFields* relation for record fields

and its projection, V_P does require a relationship between values, expressed by the relation $projedVal(e^*, e)$, shown in Figure 7.10. This constraint stated as

$$\forall e, e', fECtx, eCtx, v, o. proj^e_P(e, e') \supset evalExpr_P(fECtx, eCtx, e, v, o) \supset \exists v'. evalExpr_P(fECtx, eCtx, e', v', o) \wedge projedVal(v, v')$$

which also requires the same printed output. The relation $projedVal(e_1, e_2)$ can be described at a high level as e_1 taking zero or more projection steps somewhere throughout its structure to become equal to e_2. It allows an expression and its projection to evaluate to the same value (PV-Zero), or the projection to evaluate to the projection of the original value (PV-Proj). It also allows an expression to evaluate to a new value form that projects to a record and the expression's projection to evaluate to a record as well, but with fields containing values related by projection (PV-Trans with PV-Proj and PV-Rec). This relation between values allows extensions the freedom to introduce new value forms, but also gives extensions introducing properties some idea of what those new value forms mean.

It is not immediately clear why this projection constraint should induce us to use relations like *matchInt* and *matchRec* in defining the host language. To understand why, we need to consider an extension, one that introduces a new value form and expression forms that operate on that form of value. The new expression forms need to project to something from the host language, something that, without matching, expects the sub-expressions to evaluate to exactly a value form introduced by the host language. In V_L, we can write projections that avoid this problem but satisfy the projection constraints by evaluating to entirely unrelated values, but V_P's projection constraints do not allow this, requiring closely-related values. Defining the host language's semantics through matching enables the host language's constructs to operate on extension-introduced value forms, enabling extensions to introduce new value forms. We will discuss this further, and more concretely, in the context of the list extension, which this matching enables to introduce new value forms.

To help extensions understand the meaning of extension-introduced value forms, the host language also introduces properties requiring values related by *projedVal* to match the same value forms. This gives us properties like

$$\begin{aligned} &\forall e, e', i. \mathit{projedVal}(e, e') \supset \mathit{matchInt}(e, i) \supset \mathit{matchInt}(e', i) \\ &\forall e, e', i. \mathit{projedVal}(e, e') \supset \mathit{matchInt}(e', i) \supset \mathit{matchInt}(e, i) \end{aligned} \tag{7.7}$$

We have similar properties for each *match* relation. The effect of using matching relations and requiring them to carry through *projedVal* is similar to the pattern matching built into Silver [15, 38]. In Silver, matching a value v against a pattern $\mathit{intlit}(i)$ automatically checks if v is constructed by *intlit* and, if not, projects it and tries the matching again. This prior experience suggests that the matching approach used in V_P might be useful.

In addition to these properties, V_P has many of the same properties as V_L, but with V_P's relations replacing V_L's relations. It also requires statement evaluation to produce the same results (Property 7.4), that projections for all syntax categories be unique, that expressions' projections have the same variables (Property 7.5), and that expression evaluation be determined entirely by the values assigned to an expression's variables in the evaluation

context (Property 7.6), as well as introducing the $addP(evalExpr_P)$ and $addP(evalStmt_P)$ properties, with proxy rules like those introduced by V_L.

Exact Evaluation in V_E

Our final version of the language library, V_E, requires the evaluation of a projection to be exactly the same as that for the projecting term's evaluation.[5] This is similar to but stricter than what we saw in V_P. To aid us in highlighting the effects of changes in the host language on what extensions may introduce, we also add a new construct to the shared underlying language, one that will allow us to nest statements inside expressions.

The evaluation and projection relations for V_E's host language can be found in Figure 7.11. Because we can now nest statements inside expressions, the evaluation relation for expressions includes an updated evaluation context, the same as statement evaluation does, in addition to a value. This is threaded through rules, with each sub-expression and sub-statement using the updated evaluation context from the previously-evaluated one, as we see in E_E-Add and X_E-If-True. Our new expression form that allows us to nest statements inside expressions is $stmtExpr(s, e)$, with evaluation rule E_E-StmtExpr. We evaluate the statement s in a new scope, then evaluate the expression e in the updated evaluation context, only dropping the new scope after the expression has been evaluated, allowing it to use new variables declared by the statement. Note also that our projection relation for expressions now includes a set of names, just as the one for statements does. Because we can nest statements inside expressions, we may find it useful to generate fresh names for temporaries in projecting expressions as well as statements.

V_E introduces two projection constraints for expression evaluation. First, it requires an expression's projection to evaluate with the same printed output, as do the other language

[5]The full Sterling and Extensibella development can be found at `https://mmel.cs.umn.edu/extensibella/examples/exactEval/description.html`.

Relations include

$evalExpr_E(fECtx, eCtx, e^*, e, eCtx, output)$, $evalStmt_E(fECtx, eCtx, s^*, eCtx, output)$, $evalProg_E(\overline{e}, prog^*, output)$

Projection relations include

$proj^e_E(2^n, e^*, e)$, $proj^s_E(2^n, s^*, s)$, $proj^{prog}_E(prog^*, prog)$

$\boxed{evalExpr_E(fECtx, eCtx, e^*, e, eCtx, output)}$

$$\frac{}{evalExpr_E(fECtx, eCtx, intlit(i), intlit(i), eCtx, nilOutput)}\ \text{E}_E\text{-Intlit}$$

$$\frac{\begin{array}{c}evalExpr_E(fECtx, eCtx, e_1, intlit(i_1), eCtx_1, o_1)\\ evalExpr_E(fECtx, eCtx_1, e_2, intlit(i_2), eCtx', o_2)\\ plus(i_1, i_2, i) \qquad concat(o_1, o_2, o)\end{array}}{evalExpr_E(fECtx, eCtx, add(e_1, e_2), intlit(i), eCtx', o)}\ \text{E}_E\text{-Add}$$

$$\frac{\begin{array}{c}evalStmt_E(fECtx, addScope(nilval, eCtx), s, eCtx_1, o_1)\\ evalExpr_E(fECtx, eCtx_1, e, v, addScope(\gamma, eCtx'), o_2) \qquad concat(o_1, o_2, o)\end{array}}{evalExpr_E(fECtx, eCtx, stmtExpr(s, e), eCtx', o)}\ \text{E}_E\text{-StmtExpr}$$

$\boxed{evalStmt_E(fECtx, eCtx, s^*, eCtx, output)}$

$$\frac{\begin{array}{c}evalExpr_E(fECtx, eCtx, e, intlit(i), eCtx', o')\\ concat(o', consOutput(intlit(e), nilOutput), o)\end{array}}{evalStmt_E(fECtx, eCtx, print(e), eCtx', o)}\ \text{X}_E\text{-Print-Int}$$

$$\frac{\begin{array}{c}evalExpr_E(fECtx, eCtx, e, true, eCtx_1, o_1)\\ evalStmt_E(fECtx, addScope(nilval, eCtx_1), s_t, addScope(\gamma, eCtx'), o_2)\\ concat(o_1, o_2, o)\end{array}}{evalStmt_E(fECtx, eCtx, ifte(e, s_t, s_f), eCtx', o)}\ \text{X}_E\text{-If-True}$$

Figure 7.11: Selected relations and rules in V_E's host language

libraries, but also the same value and the same updated evaluation context:

$$\forall ns, e, e', fECtx, eCtx, v, eCtx', o.proj^{e}_{E}(ns, e, e') \supset namesOf(eCtx, ns) \supset$$
$$evalExpr_{E}(fECtx, eCtx, e, v, eCtx', o) \supset evalExpr_{E}(fECtx, eCtx, e', v, eCtx', o)$$

The second one requires an expression to evaluate whenever its projection does:

$$\forall ns, e, e', fECtx, eCtx, v, eCtx', o.proj^{e}_{E}(ns, e, e') \supset namesOf(eCtx, ns) \supset$$
$$evalExpr_{E}(fECtx, eCtx, e', v, eCtx', o) \supset evalExpr_{E}(fECtx, eCtx, e, v, eCtx', o)$$

Taken together, these require the evaluation behavior of extension-introduced constructs to be exactly the same as their projections, so extensions proving properties can understand the behavior of other extensions very well.

Statements are similarly restricted to require exactly the same behavior for extension-introduced constructs and their projections. If a statement evaluates, its projection evaluates with the same updated context and printed output:

$$\forall ns, s, s', fECtx, eCtx, eCtx', o.proj^{s}_{E}(ns, s, s') \supset namesOf(eCtx, ns) \supset$$
$$evalExpr_{E}(fECtx, eCtx, s, eCtx', o) \supset evalExpr_{E}(fECtx, eCtx, s', eCtx', o)$$

If a statement's projection evaluates, so does the projecting statement:

$$\forall ns, s, s', fECtx, eCtx, eCtx', o.proj^{s}_{E}(ns, s, s') \supset namesOf(eCtx, ns) \supset$$
$$evalExpr_{E}(fECtx, eCtx, s', eCtx', o) \supset evalExpr_{E}(fECtx, eCtx, s, eCtx', o)$$

These four projection constraints give extensions a very strong foundation for proving new properties about evaluation, as we shall see in discussing the extensions written for V_E.

Finally, as do the other two language libraries, V_E introduces the $addP(evalExpr_E)$ and $addP(evalStmt_E)$ properties, once again with similar proxy rules, to allow extensions to

introduce properties using these evaluation relations as their key relations.

We see that, while there are differences in the exact definitions of the host languages of the different libraries in addition to their projection constraints, overall they define essentially the same underlying language, other than V_E's addition of the *stmtExpr* form. Note, however, that this form does not increase the power of V_E relative to the others; any program written using it can be rewritten to avoid it. We shall see in the remainder of this section how the differences in the definitions of the host languages, despite appearing relatively minor, and the changes to the projection constraints are significant to what we can and cannot write in extensions.

7.4.2 Syntactic Extensions

We start our examination of the different capabilities of the different libraries with the syntactic extensions to the language. The first two extensions are the ascription and assertion extensions, with rules for V_L shown in Figure 7.12; the evaluation and projection rules for the versions of the extensions in the other language libraries are similar. The ascription extension adds an expression form $\mathit{ascribe}(e, ty)$ that wraps another expression, expecting it to have the given type; this expression form is useful in programming to help one fix type errors. It defines typing to check e has type ty, but otherwise treats itself as if it were only e, and projects to e.

The second extension adds a statement form $\mathit{assert}(e)$ that asserts e evaluates to *true*, abruptly terminating the program if that is not the case. Assertion statements are useful for detecting bugs, showing when one's assumptions about how values should be related are false. The *assert* construct projects to an if-then-else that evaluates the expression, doing nothing else if it is true, and abruptly terminating the program using the host language's *error* construct if the expression is false. The simplicity of these extensions, and the fact their meanings so directly match what is available in the host language, allows all three language libraries to accept them with the same rules and projections.

$$\frac{evalExpr_L(fECtx, eCtx, e, v, o)}{evalExpr_L(fECtx, eCtx, ascribe(e, ty), v, o)}\ \text{E}_L\text{-Ascribe} \qquad \frac{}{proj^e_L(ascribe(e), e)}\ \text{Proj}_L\text{-Ascribe}$$

$$\frac{evalExpr_L(fECtx, eCtx, e, true, o)}{evalStmt_L(fECtx, eCtx, assert(e), eCtx', o)}\ \text{X}_L\text{-Assert}$$

$$\frac{}{proj^s_L(assert(e), ifte(e, skip, print(error)))}\ \text{Proj}_L\text{-Assert}$$

Figure 7.12: Evaluation and projection rules for the ascription and assertion extensions

$$\frac{evalExpr_L(fECtx, eCtx, c, true, o_1) \quad evalExpr_L(fECtx, eCtx, t, v, o_2) \quad concat(o_1, o_2, o)}{evalExpr_L(fECtx, eCtx, condExpr(c, t, f), v, o)}\ \text{E}_L\text{-CondExpr-True}$$

$$\frac{evalExpr_L(fECtx, eCtx, c, false, o_1) \quad evalExpr_L(fECtx, eCtx, t, v, o_2) \quad concat(o_1, o_2, o)}{evalExpr_L(fECtx, eCtx, condExpr(c, t, f), v, o)}\ \text{E}_L\text{-CondExpr-False}$$

Figure 7.13: Evaluation rules for *condExpr* in V_L

Conditional Expression

The next extension, the one introducing a conditional expression, is where the libraries start diverging. The new conditional expression form $condExpr(e, e, e)$, commonly called the *ternary expression* in languages like C and Java, takes conditional evaluation to the expression level. Depending on whether the first sub-expression evaluates to *true* or *false*, its result is either the result of evaluating the second or third sub-expressions. Its evaluation rules from V_L are found in Figure 7.13. Unlike the previous two extensions, this construct is trying to bring something new, arbitrary branching, to the expression level.

Recall that V_L requires expression projections to evaluate and have the same printed output. This initially appears problematic for introducing $condExpr(c, t, f)$, as V_L's host

language does not include explicit branching, and we need branching to evaluate *either* t or f and get its printed output, but not both. However, V_L's projection constraint also allows extensions to evaluate to unrelated values, so we can use the *implicit* branching of short-circuiting boolean operations, projecting $\mathit{condExpr}(c, t, f)$ to

$$\mathit{or}(\mathit{and}(c, \mathit{or}(\mathit{eq}(t, \mathit{intlit}(0)), \mathit{true})), \mathit{eq}(f, \mathit{intlit}(0)))$$

This always evaluates c, as the conditional expression does, so the printed output of the whole includes c's printed output. If c evaluates to *true*, the *and* continues, which evaluates t. Since the second conjunct always evaluates to *true*, we then do not evaluate f, so the expression's printed output is the same as in the E_L-CondExpr-True rule it is emulating. If c evaluates to *falseVal*, we go and evaluate f, as expected, giving the same printed output as in the E_L-CondExpr-False rule it is emulating. Thus the projection constraint is satisfied, even though the values will only ever be related by random chance.

The version of the extension for V_E has similar evaluation rules to those found in Figure 7.13, but using $\mathit{evalExpr}_E$ instead and threading through context updates. Its language library requires the projection to produce not only the same printed output, but also the same value. To do so, it can wrap a conditional *statement* in a *stmtExpr* to accomplish the branching:

$$\mathit{stmtExpr}(\mathit{seq}(\mathit{decl}(x, \mathit{intTy}, \mathit{intlit}(0)), \mathit{ifte}(c, \mathit{assign}(x, t), \mathit{assign}(x, f))), x)$$

In this, x is a fresh name generated using *freshName* to use as a temporary. The use of *intTy* and $\mathit{intlit}(0)$ in the declaration of x are simply placeholders; we do not need the type to match the type of t and f, and we will replace the initial value without using it. It is clear that this has exactly the same behavior as the conditional expression has, conditionally assigning the correct value to x, which is then the value in which its evaluation results. It is also clear that whenever one of the term or its projection evaluates, the other will as well.

Most interesting of all, perhaps, is that V_P cannot support this extension. In V_P,

projections need to evaluate to values related to those of the projecting expressions, so the boolean short-circuiting evaluation approach of V_L does not work. Because V_P's host language does not include *stmtExpr* like V_E's host language does, it cannot use statement-level branching to accomplish this either. This shows how the richness of the host language and the strictness of the projection constraints in a language library both affect what extensions may introduce. If a language library has strict projection constraints but a rich host language, as in V_E, it can support a lot of extensions with interesting semantics because the projections can match the extension behavior. Similarly, if a language library has a poorer host language but loose projection constraints, such as V_L, it can also support a lot of extensions because the projections do not need to match the extension behavior closely. However, when we have somewhat stricter projection constraints without enriching the language, as in V_P, there are some extensions we cannot write. This is true even when the same computations can be represented by both the richer and poorer host languages, as we see here, because of *how* they can be represented.

List Extension

Our final syntactic extension is the list extension, adding new forms for expressions, statements, values, and types. The full set of constructs the list extension seeks to add include all those introduced by the list extension in Chapter 2, as well as new expression forms $\mathit{index}(e, e)$ for accessing an element in a list at a certain index and $\mathit{length}(e)$ for taking the length of a list. The definitions of the semantics for the expression forms are as expected. They also include two new statement forms. The first, $\mathit{listUpdate}(n, e, e)$, updates the element in a list at a given index, with $\mathit{listUpdate}(l, i, e)$ being similar to a Python list update or Java array update `l[i] = e`. The other statement form, $\mathit{listForeach}(n, e, s)$, executes the loop's body s once for each element of the list e, with the name n being associated with the current element, similar to a `foreach` loop in Java or a `for` loop over a list in Python. The rules for these new statement constructs for V_L, along with the auxiliary relations that help implement them, are shown in Figure 7.14, with the definitions from the extensions to the other language libraries implementing the same behavior. The constructs each version

$\boxed{evalStmt_L(fECtx, eCtx, s^*, eCtx, output)}$

$$\frac{\begin{array}{c} lkpVal(eCtx, l, lv) \qquad evalExpr_L(fECtx, eCtx, idx, intlit(i), o_1) \\ evalExpr_L(fECtx, eCtx, e, v, o_2) \qquad updateListIndex_L(lv, i, v, lv') \\ update(eCtx, l, lv', eCtx') \qquad concat(o_1, o_2, o) \end{array}}{evalStmt_L(fECtx, eCtx, listUpdate(l, idx, e), eCtx', o)} \text{ X}_L\text{-ListUpdate}$$

$$\frac{\begin{array}{c} evalExpr_L(fECtx, eCtx, e, lv, o_1) \\ iterate_L(fECtx, eCtx, lv, n, s, eCtx', o_2) \qquad concat(o_1, o_2, o) \end{array}}{evalStmt_L(fECtx, eCtx, listForeach(n, e, s), eCtx', o)} \text{ X}_L\text{-ListForeach}$$

$\boxed{updateListIndex_L(e, i^*, e, e)}$

$$\frac{}{updateListIndex_L(cons(hd, tl), 0, v, cons(v, tl))} \text{ ULI-0}$$

$$\frac{subtract(i, 1, i') \qquad updateListIndex_L(tl, i', v, tl')}{updateListIndex_L(cons(hd, tl), i, v, cons(hd, tl'))} \text{ ULI-Step}$$

$\boxed{iterate_L(fECtx, eCtx, e^*, n, s, eCtx, output)}$

$$\frac{}{iterate_L(fECtx, eCtx, nil, n, s, eCtx, nilOutput)} \text{ I}_L\text{-Nil}$$

$$\frac{\begin{array}{c} evalStmt_L(fECtx, addScope(consval(n, hd, nilval), eCtx), s, addScope(\gamma, eCtx_1), o_1) \\ iterate_L(fECtx, eCtx_1, tl, n, s, eCtx', o_2) \qquad concat(o_1, o_2, o) \end{array}}{iterate_L(fECtx, eCtx, cons(hd, tl), n, s, eCtx', o)} \text{ I}_L\text{-Cons}$$

Figure 7.14: Rules for evaluating list statements in V_L's list extension

of the extension successfully adds to each version of the language library are shown in Table 7.2, along with those expression forms it identifies as values. As we will discuss below, V_P is unable to support some of the expression forms for reasons similar to its inability to support the conditional expression, and V_E does not identify new value forms due to its projection constraints for expression evaluation.

We start our discussions of the extensions in the different language libraries with V_L, which is the simplest of the three. This version of the extension includes all the constructs the extension designer seeks to add, as shown in Table 7.2. The projections of the list

	V_L	V_P	V_E
Expression Forms	nil $cons$ $head(e)$ $tail(e)$ $null(e)$ $index(e, e)$ $length(e)$	nil $cons$ $head(e)$ $tail(e)$ $null(e)$	nil $cons$ $head(e)$ $tail(e)$ $null(e)$ $index(e, e)$ $length(e)$
Statement Forms	$listUpdate(n, e, e)$ $listForeach(n, e, s)$	$listUpdate(n, e, e)$ $listForeach(n, e, s)$	$listUpdate(n, e, e)$ $listForeach(n, e, s)$
Type Forms	$listTy(ty)$	$listTy(ty)$	$listTy(ty)$
Forms Identified as Values	nil $cons(e, e)$	nil $cons(e, e)$	

Table 7.2: New constructors introduced by the list extension for each version of the language library

expressions are generally the same as those in the list extension in Chapter 2, projections like $head(e)$ projecting to e and $cons(e_1, e_2)$ projecting to $eq(e_1, e_2)$; these sorts of projections are extended to the two new expression forms as well, matching the printed output of the list expressions as required by the projection constraint. To match the stricter constraint for statements, the two new statement forms need to have more exact, and therefore more complex, projections. These are shown in Figure 7.15 using program fragments written in a Java-like language for clarity; these fragments map to statements in our host language with the list extension, but they are nearly unreadable when written using abstract syntax alone. The projection of *listUpdate* saves the relevant values in new temporaries, then pulls the elements before the index off the list and stores them, replaces the correct index when it is reached, then piles the other elements back onto the list. This matches the behavior we see in the X_L-ListUpdate, ULI-0, and ULI-Step rules in Figure 7.14. The projection of *listForeach* creates a *while* loop to execute the body for each element of the list, just as we see in the X_L-ListForeach, I_L-Nil, and I_L-Cons rules. It is clear upon examination that both of these projections have the same effects as the statements that project to them, satisfying our constraints.

V_P supports most of the constructs, including the two statement forms that use the same evaluation rules and projections as V_L, but not the *index* or *length* expressions. Both list

$proj^s_L(ns, listUpdate(l, i, e), s')$
where s' is

```
{
   intTy SaveI = i;
   intTy SaveE = e;
   intTy Hold = [];
   while (SaveI > 0){
      SaveI = SaveI - 1;
      Hold = head(l)::Hold;
      l = tail(l);
   }
   l = SaveE::tail(l);
   while (!null(Hold)){
      l = head(Hold)::l;
      Hold = tail(Hold);
   }
}
```

where `SaveI`, `SaveE`, `Hold` are unique fresh names relative to $ns \cup \{l\}$

$proj^s_L(ns, listForeach(x, l, s), s')$
where s' is

```
{
   intTy SaveL = l;
   while (!null(SaveL)){
      intTy x = head(SaveL);
      SaveL = tail(SaveL);
      s
   }
}
```

where `SaveL` is a fresh name relative to $ns \cup \{x\}$

N.B. The outer set of braces on both denote opening a new scope (*scopeStmt*)

Figure 7.15: List statement projections in V_L, which match those in V_P

indexing and list length depend on traversing an arbitrarily-sized portion of the list, requiring looping, which V_P's host language does not support at the expression level. However, the inability of V_P to introduce these as new expression constructs does not mean programs in a language composed from V_P including its list extension cannot find the length of a list or an element at an index. Statements carrying out these operations are expressible using constructs from V_P's host language and list extension. It is simply in representing them as *expressions*, and in introducing expression constructs carrying them out, that V_P is lacking.

The list expressions that are present in V_P project to record expressions, a choice that is not only natural, but forced by the projection constraint they must obey. List expressions such as $head(e)$ and $tail(e)$ need to project to expressions that evaluate to values related by *projedVal*. To do so, we need expression forms from the host language that can operate on the list value $cons(hd, tl)$ to which the sub-expressions of both $head(e)$ and $tail(e)$ must evaluate, with one projection choosing hd and one choosing tl. Because the projection of the list value needs to maintain both the head and the tail values to fulfill the projection

constraint for the evaluation of the *head*(e) and *tail*(e) constructs, it must project to a record value, the only structured value the host language offers. We can project *cons*(hd, tl) to a record value with a *head* field with the value *hd* and a *tail* field with the value *tl*, with *head*(e) and *tail*(e) projecting to record accesses of the appropriate fields. Because the record field accesses are defined using the *matchRec* relation, they can operate on *cons*(hd, tl) as if it were the record value to which it projects. To support the *null*(e) construct, the record value to which a *cons* value projects also contains a *null* field with the value *false*.

Our final language library version, V_E, similarly projects list expressions to records, but its definitions end up rather complicated and unintuitive due to its projection constraints requiring exact matching of evaluation results. This constraint means its constructs cannot use new values. This is why V_E does not identify *nil* or *cons*(e, e) as value forms like the list extensions in V_L and V_P do; it could identify them as such, but it could not use them as such. Rather, as in V_P, our list expressions project to record expressions, and, to satisfy the projection constraints, list expressions instead evaluate to and operate on record values, as we see in Figure 7.16. An expression *nil* evaluates to a record value with a *null* field, and *null*(e) accesses this *null* field to determine whether a "list" value is null or not. Comparing the rules for evaluation of the list statements in V_L's list extension, shown in Figure 7.14, and the ones for V_E in Figure 7.16, we see the use of record values for expressions also makes their definitions more complex, as they must now operate on record values as well. V_E can also support the list extension's *index* and *length* expressions by projecting them to loops wrapped in *stmtExpr*, similar to how it projected *condExpr* to an if-then-else statement.

Recall we also expect bidirectional existence of statement evaluation, and matching of results across projection. This is simple for the *listForeach* construct, which can use the same projection as in V_L and V_P. However, the projection of *listUpdate* must become more complex to support the evaluation of the original statement existing whenever the projection evaluates. The new projection is shown in Figure 7.17; compare this to the projection in Figure 7.15. The reason the new projection is more complex is that we can't count on the "list" value we are updating actually being list-shaped, even counting a record with *null*, *head*, and *tail* fields as list-shaped; it could have extra fields, or, without some of the checks

$\boxed{evalExpr_E(fECtx, eCtx, e^*, e, eCtx, output)}$

$$\frac{}{evalExpr_E(fECtx, eCtx, nil, rec(consFs(null, true, nilFs)), eCtx, nilOutput)}\ \text{E}_E\text{-Nil}$$

$$\frac{evalExpr_E(fECtx, eCtx, e, rec(fs), eCtx', o) \qquad lookupField(fs, null, v)}{evalExpr_E(fECtx, eCtx, null(e), v, eCtx, o)}\ \text{E}_E\text{-Null}$$

$\boxed{evalStmt_E(fECtx, eCtx, s^*, eCtx, output)}$

$$\frac{\begin{array}{c} lkpVal(eCtx, l, lv) \qquad evalExpr_E(fECtx, eCtx, idx, intlit(i), o_1) \\ evalExpr_E(fECtx, eCtx, e, v, o_2) \qquad updateListIndex_E(lv, i, v, lv') \\ update(eCtx, l, lv', eCtx') \qquad concat(o_1, o_2, o) \end{array}}{evalStmt_E(fECtx, eCtx, listUpdate(l, idx, e), eCtx', o)}\ \text{X}_E\text{-ListUpdate}$$

$$\frac{\begin{array}{c} evalExpr_E(fECtx, eCtx, e, lv, o_1) \\ iterate_L(fECtx, eCtx, lv, n, s, eCtx', o_2) \qquad concat(o_1, o_2, o) \end{array}}{evalStmt_E(fECtx, eCtx, listForeach(n, e, s), eCtx', o)}\ \text{X}_E\text{-ListForeach}$$

$\boxed{updateListIndex_E(e, i^*, e, e)}$

$$\frac{lookupField(fs, null, false) \qquad replaceField(fs, head, v, fs')}{updateListIndex_E(rec(fs), 0, v, rec(fs))}\ \text{ULI-0}$$

$$\frac{\begin{array}{c} lookupField(fs, null, false) \qquad subtract(i, 1, i') \qquad lookupField(fs, tail, tl) \\ updateListIndex_E(tl, i', v, tl') \qquad replaceField(fs, tail, tl', fs') \end{array}}{updateListIndex_E(rec(fs), i, v, rec(fs'))}\ \text{ULI-Step}$$

$\boxed{iterate_E(fECtx, eCtx, e^*, n, s, eCtx, output)}$

$$\frac{lookupField(fs, null, true)}{iterate_E(fECtx, eCtx, rec(fs), n, s, eCtx, nilOutput)}\ \text{I}_E\text{-Nil}$$

$$\frac{\begin{array}{c} lookupField(fs, null, false) \qquad lookupField(fs, head, head) \\ evalStmt_E(fECtx, addScope(consval(n, hd, nilval), eCtx), s, addScope(\gamma, eCtx_1), o_1) \\ lookupField(fs, tail, tl) \\ iterate_E(fECtx, eCtx_1, tl, n, s, eCtx', o_2) \qquad concat(o_1, o_2, o) \end{array}}{iterate_E(fECtx, eCtx, rec(fs), , n, s, eCtx', o)}\ \text{I}_E\text{-Cons}$$

Figure 7.16: Selected evaluation rules for the list extension in V_E

$proj_s(ns, listUpdate(l, i, e), s')$ where s' is

```
{
   intTy SaveI = i;
   intTy SaveE = e;
   intTy Hold = [];
   while (SaveI != 0 && !null(l)){
      SaveI = SaveI - 1;
      intTy Copy = l;
      Copy.tail = Hold;
      Hold = Copy;
      l = tail(l);
   }
   if (null(l)){
      intTy Copy = error;
   }
   else {}
   l = SaveE::tail(l);
   while (!null(Hold)){
      intTy Copy = Hold;
      Copy.tail = l;
      l = Copy;
      Hold = tail(Hold);
   }
}
```

where `SaveI`, `SaveE`, `Hold`, and `Copy` are unique fresh names relative to $ns \cup \{l\}$

Figure 7.17: Projection for *listUpdate* in V_E

added, not enough fields. Using a temporary variable `Copy` we can use the same intuitive approach as before, but still save any extra fields that the $listUpdate_E$ relation would leave unmodified. This projection allows us to prove the evaluation of the *listUpdate* and its projection are exactly the same, as this version of the language requires.

In considering our syntactic extensions in the three language libraries, we see how both the syntax and projection constraints from a library's host language affect the syntax extensions introduce and how they extend the definitions of the existing semantics to their new syntax. With V_P, we saw how relatively strict projection constraints and a relatively poor host language can combine to limit what semantics extensions may introduce relative to languages with either richer host languages or less-restrictive projection constraints. This suggests that the constructs and semantics extensions may introduce are not necessarily

hampered by choices of projection constraints or constructs included in the host language, so long as they are appropriately coordinated with one another.

We also see how projection constraints can affect exactly how the semantics of constructs are defined. Even though the list extensions of V_L, V_P, and V_E show programmers the same behavior, other than V_P missing some constructs, the definitions of those semantics in V_E are complicated by the requirement that its list expressions evaluate to the record values their projections do. In contrast to the more standard definitions enabled by the looser projection constraints of V_L and V_P, it is more difficult to read and reason about the definitions given by V_E. This strictness also requires a more complex projection for *listUpdate*; even having written the other two libraries' projections and proven them correct, getting insight into what the projection needed, the author had to iterate on the definitions of the projection and the $updateListIndex_E$ relation to be able to prove V_E's projection constraints for them due to the difficulties introduced by arbitrary record values.

While the strictness of projection constraints can negatively affect the freedom of extensions in introducing new syntax and the ease of extending existing semantic relations to them, strictness can be a benefit for extensions proving new properties, as we shall see with our semantic extensions.

7.4.3 Security Extension

The first semantic extension we consider is for information flow security, as we had in Chapter 2. In addition to the *secdecl* and security level constructors it introduced before, the extension now introduces new constructors allowing functions to declare the security level of their returned values and parameters. The security analysis is extended to handle function declarations and programs, as well as the new expression and statement forms. The most important addition to note is that printing is allowed only in public contexts; printing in a private context would leak information about the values of private variables in conditions that let us reach the private context. Note also that functions are considered secure if their bodies are secure when the parameters are assigned the levels the programmer gives them.

The main function in a program can also have both public and private arguments, just as any other function can, an idea that may sound a bit strange at first. While the arguments to the main function can stand for arguments given to a program run on the terminal, as we expect, they can also stand for other inputs to it. For example, the contents of a file containing an encryption key that is used in a program might be considered an argument to the program, and thus to the main function, and a private one at that.

The main security property we want for all three versions of the language, then, is that a secure program has the same printed output if the public arguments to the main function are the same, even though the private arguments may differ. If $progSecure_L$ is the security analysis relation for V_L's security extension, relating a program to the security levels for arguments to its main function, and *eqpublicargs* holds when the public arguments in two sets have the same values, this property can be stated as

$$\forall p, s_a, a_1, a_2, o_1, o_2. progSecure_L(p, s_a) \supset eqpublicargs(s_a, a_1, a_2) \supset$$
$$evalProg_L(a_1, p, o_1) \supset evalProg_L(a_2, p, o_2) \supset o_1 = o_2$$

for V_L. The statements would be equivalent for the other two language libraries. Since the printed output is the only evaluation result we see from executing a program, it makes sense that that is our measure of leakage. Comparing this property to Property 3.3 that we had for statements in the language from Chapter 2, we see the *eqpublicargs* relation between the arguments corresponds to the *eqpublicvals* relation between contexts that requires all variables assigned public in the security context to have the same values in two evaluation contexts.

To prove the property we want about programs, we need to know properties about both statements and expressions being evaluated under evaluation contexts containing the same values for public variables but possibly different ones for private variables. The statement one is similar to Property 3.3, that public variables in the updated contexts for two evaluations are always the same, but with the addition that the printed output for both is always the same as well. The proof of this property in all three versions of the language library

proceeds much as the one in the more limited language did.

For expressions, we need to know that the printed output for two evaluations under related contexts is the same, but also that expressions with a public security level produce the same values when evaluated under both evaluation contexts. Due to the stricter constraints V_E requires of extensions versus the other two libraries, we can provide a proof of this property directly in V_E, while we must take a different approach for the other two.

In V_E, we can prove these properties with expression evaluation as the key relation. In the generic case we know the security $level_E$ relation is defined by projecting the generic expression to an expression e' and finding the level of e'. We also know that an expression and its projection evaluate to exactly the same value and have exactly the same printed output. Thus we can use the induction hypothesis with the evaluation and $level_E$ derivation for e' to show both of its evaluations produce the same value and output, then directly lift back the result to the generic expression, completing the generic case. As all the known cases are direct, the proof is complete.

We cannot use this approach for V_L and V_P because we don't have the same value produced by the projection as its original expression. Even in V_P, where the values are related by *projedVal*, we cannot lift the property back from the projection like this. What we know is that our two evaluations of the generic constructor produce values v_1 and v_2 under the related evaluation contexts, that the projection e' produces a single value v under both, and that we have $projedVal(v_1, v)$ and $projedVal(v_2, v)$. This does not tell us that v_1 and v_2 are the same, as we would need in this approach to the proof, because several values could be related to v. For example, if v were a record value, v_1 could be another record value and v_2 could be a list value. The failure of this approach is even clearer for V_L, as we have no relationship between the evaluation results of the generic constructor and its projection e'.

Instead, we can prove an expression with level *public* contains only variables assigned *public* in the security context, with the $level_L$ or $level_P$ relation as the key relation. In the generic case, the proof relies on the projection constraint that an expression's variables are a subset of its projection's variables (Property 7.5), which both versions of the language

library have. We can then use the host language's property that expression evaluation is determined by the values assigned to its variables in the evaluation context (Property 7.6), also shared by both libraries, and the fact that the two evaluation contexts have all the same values for public variables, to show the values produced by any expression under the two related evaluation contexts is the same.

Thus we have the same fact for expressions, which supports the properties for statements and ultimately for programs, in all three libraries, but by two different approaches. The fact we use different approaches to reach the same conclusions here does not affect programmers. A programmer using a composed language built from any one of these three language libraries sees the same thing, a language where she declares security levels for functions and variables and receives a static guarantee information marked as confidential cannot be leaked by any execution of the program.

7.4.4 Translation Extension

Our final extension is for translating the language to itself, or, more specifically, to the host language. This self-translation might seem to be a strange choice, but it makes a good stand-in for considering the limitations on translating to any language, or any sort of program transformation an extension might introduce. By testing a self-translation, we keep the transformation simple, making both the rules implementing it and the proofs of properties relatively simple as well.

The translation extension is defined in the same way for all three versions of the language, with a rule for each constructor that simply translates the sub-terms. Some of V_L's rules for translating expressions can be found in Figure 7.18, with the rules being equivalent for V_P and V_E. In TE_L-Add, we see an addition $add(e_1, e_2)$ translates e_1 to e_1' and e_2 to e_2', with the overall translation being $add(e_1', e_2')$. Other rules follow this same pattern, both for expressions and for statements. Note the statement relation includes both an incoming set of used names and an outgoing, updated one for new declarations, this set of names being used in the default rule. The default rules for both expressions (TE_L-Default) and statements (TS_L-Default) project the term, with statement projections using the set of

$\boxed{transE_L(e^*, e)}$

$$\frac{}{transE_L(intlit(i), intlit(i))}\ \text{TE}_L\text{-Int} \qquad \frac{}{transE_L(var(n), var(n))}\ \text{TE}_L\text{-Var}$$

$$\frac{transE_L(e_1, e_1') \qquad transE_L(e_2, e_2')}{transE_L(add(e_1, e_2), add(e_1', e_2'))}\ \text{TE}_L\text{-Add}$$

$\boxed{transS_L(2^n, s^*, s, 2^n)}$

$$\frac{transS_L(ns, s_1, s_1', ns_1) \qquad transS_L(ns_1, s_2, s_2', ns')}{transS_L(ns, seq(s_1, s_2), seq(s_1', s_2'), ns')}\ \text{TS}_L\text{-Seq}$$

$$\frac{transE_L(e, e')}{transS_L(ns, assign(n, e), assign(n, e'), ns)}\ \text{TS}_L\text{-Assign}$$

Default rules ($\mathbb{S}$):

$$\frac{proj_L^e(e, e_p) \qquad transE_L(e_p, e')}{transE_L(e, e')}\ \text{TE}_L\text{-Default}$$

$$\frac{proj_L^s(ns, s, s_p) \qquad transS_L(ns, s_p, s', ns')}{transS_L(ns, s, s', ns')}\ \text{TS}_L\text{-Default}$$

Figure 7.18: Selected rules for V_L's expression translation relations

used names to generate fresh ones if needed, taking the translation of its projection as its translation.

A relation implementing a translation such as this might be a way to make a runnable implementation of a language. A program in the extensible language is translated to one in a more common programming language, which is then compiled and executed. For example, the Silver attribute grammar system [38], itself an extensible language, is implemented by translation to Java code that can be compiled and executed.

The correctness of such a translation can be specified by the evaluation of the translated program having behavior related in certain ways to the original program. What we can prove about the translations in our different language libraries, which are all defined by the same rules, differs greatly based on the projection constraints each one provides.

For V_E, our projection constraints let us prove the evaluation of the translation is exactly the same as the original, that is, we can prove

$$\forall s, s', ns, ns', fECtx, fECtx', eCtx, eCtx', o.$$
$$evalStmt_E(fECtx, eCtx, s, eCtx', o) \supset transS_E(ns, s, s', ns') \supset$$
$$transRelFuns_E(fECtx, fECtx') \supset evalStmt_E(fECtx', eCtx, s', eCtx', o)$$

where $transRelFuns_E$ relates two function contexts where the functions in the second are translations of the functions in the first, that a statement's translation evaluates with the same results as the original statement. We can also prove

$$\forall s, s', ns, ns', fECtx, fECtx', eCtx, eCtx', o.$$
$$evalStmt_E(fECtx', eCtx, s', eCtx', o) \supset transS_E(ns, s, s', ns') \supset$$
$$transRelFuns_E(fECtx, fECtx') \supset evalStmt_E(fECtx, eCtx, s, eCtx', o)$$

that a statement evaluates with the same results when its translation does. We can prove similar properties for expressions and programs. These proofs are possible because, in the generic cases, we know the translation is defined by the default rule, which translates the generic term's projection. The projection constraints tell us a statement's projection evaluates if the original statement does, and vice versa, and the induction hypothesis tells us the same about the projection and its translation. Then, transitively, the projection's translation, which is also the original statement's translation, evaluates if and only if the original statement evaluates as well, and with the same results. Because the evaluation behavior of a program and its translation is exactly the same, using a translation as an implementation seems like a reasonable choice for V_E.

In V_P, we can prove the translation of an expression evaluates to a value to which the original's value is related by *projedVal* when the values in the contexts for the same names

are also related by *projedVal* (relation *projedCtxs*):

$$\forall e, e', fECtx, fECtx', eCtx, eCtx', v, o. evalExpr_P(fECtx, eCtx, e, v, o) \supset transE_P(e, e') \supset$$
$$transRelFuns_P(fECtx, fECtx') \supset projedCtxs(eCtx, eCtx') \supset$$
$$\exists v'. evalExpr_P(fECtx', eCtx', e', v', o) \wedge projedVal(v, v')$$

We have similar properties for statements and programs. In the known, non-generic cases for this property, we use the properties introduced by the host language requiring values related by *projedVal* to match the same value forms, such as that if we have $projedVal(v, v')$ and $matchInt(v, i)$, we also have $matchInt(v', i)$. For example, in the case of $add(e_1, e_2)$, which translates to $add(e_1', e_2')$, we know e_1 evaluates to v_1 and $matchInt(v_1, i_1)$ because $add(e_1, e_2)$ evaluates using the E_P-Add rule. The induction hypothesis then shows e_1' evaluates to a value v_1' and $projedVal(v_1, v_1')$. Then we also know $matchInt(v_1', i_1)$ by Property 7.7, and a similar chain of reasoning applies to the evaluation of e_2. Since both new values match integers, we can apply the E_P-Add rule for the translation, getting the same value. In the generic case, we rely on the projection constraint that an expression's projection evaluates to a value related to the original expression's value by *projedVal*. The translation of the generic expression is the translation of its projection, as defined by the default rule, we know the projection evaluates, and the induction hypothesis tells us its translation evaluates with a value related to the projection's value by *projedVal*. Then, since *projedVal* is defined as transitive (rule PV-Trans in Figure 7.10), the translation's value is related to the original expression's value.

The properties V_P can prove tell us a translated program's printed output is the same as the original program's printed output, and that the translated program evaluates whenever the original program does. However, it may be that the translated program evaluates when the original would not. Consider the list extension's constructs. The expression $head(e)$ translates to a record access $access(e', head)$ using the default rule for translation, where e translates to e'. Here e' evaluates to a record value $rec(fs)$ when the field access evaluates, but $head(e)$ will only evaluate if the value v produced by e is a list-shaped value.

Unfortunately, all we know is $projedVal(v, rec(fs))$, which does not guarantee $head(e)$ will evaluate, as v might be a record value as well. Thus we only have one direction of the correspondence for V_P. This might still make using a translation as an implementation reasonable in some situations, but the utility is not as clear as for V_E.

Finally, for V_L, we cannot prove anything about the relationship between the evaluation of the original program and its translation. We cannot prove the printed outputs are related, or that the translation's evaluation exists even if the original program evaluates. Initially it appears we should have a similar result as in V_P, where the translation evaluates if the original does, because our projection constraint also requires an expression's projection to evaluate if the projecting expression evaluates. However, the lack of a relationship between the actual values makes it untrue. Consider an expression

$$add(condExpr(eq(x, intlit(1)), intlit(3), intlit(4)), intlit(1))$$

This evaluates to either 4 or 5, depending on the value of x. However, recall from earlier that the projection of the *condExpr* is built by boolean conjunction and disjunction and evaluates to a boolean value, and its translation that comes from this projection will then also be built by boolean constructs and evaluate to a boolean value, if it evaluates at all. Then, while the projection of an individual expression evaluates, a full-program translation that uses them may not. Using a translation for anything in V_L is worthless; the existence of the translation extension here is a pipe dream.

An obvious question to ask is why all three libraries could support the same security property for programs, while the translation properties each provides are so different. It appears to be due to the differences in the expectations of the properties introduced by the two extensions. While the security extension relies on two evaluations of the *same* expression having the same value, the translation extension relies on an expression and its translation, two *different* expressions, having related values. Thus the increased strictness of the projection constraints of V_E relative to V_P, and of V_P relative to V_L, allow us to prove more for the translation extensions, where the different expressions are related by

projection, while the properties about an expression's evaluation being uniquely determined by its variables are sufficient for V_L and V_P to prove their security properties.

In considering these three language libraries with different versions of the underlying language and its metatheory, we have seen how the various choices in the constructs to include in the host language and the projection constraints affect how we write extensions, both with introducing new syntax and the properties we can prove. In the syntactic extensions, we saw how the projections and behind-the-scenes implementations of the same constructs vary from library to library. However, even though the definitions have significant differences, the behavior a programmer will see out of the same constructs from the extensions in different language libraries is the same in all cases. Even in V_E where lists evaluate directly to records, the programmer can think about a program as using lists, and the evaluation will appear the same as if the program were using list values. Thus the differences in the languages, as far as programmers using them are concerned, are almost entirely in the realm of what the languages may offer them, both in the constructs they can provide and the properties they can prove. The specifics of the behind-the-scenes choices of projection constraints matter only to the language developers who will use them to offer the publicly-visible constructs and properties to their users.

7.5 Considerations for Extensibility

As we have seen, writing extensible languages and their properties requires more thought than non-extensible languages and their properties. In addition to the considerations in the monolithic setting, one must also consider how choices will affect extensions written by others. Here we present some of the issues to be considered when writing a module, and its properties, for an extensible language.

7.5.1 Designing Language Modules

One consideration in designing language modules in light of extensibility is considering whether the abstract syntax in a language makes it difficult for extensions to work together.

For example, the security extension from Chapter 2 introduces a statement constructor *secdecl* to declare a new variable with an explicit security level. Suppose we added such an extension to a language with division, and another extension to check we do not divide by zero. This might add a *nonzerodecl* constructor to declare a new variable as holding a non-zero number. In a composed language containing both of these extensions, we cannot declare one variable both as non-zero and with an explicit security level because the declaration forms are different. When a syntax constructor contains a way for declaring information that affects a check of the rest of the program, it can be useful to allow that information to be extended so two extensions can add to it at once. This type of variation point within syntax can make extension inter-operation feasible. We see this with declarations, where we are expected to declare a type for a variable. Rather than each extension adding its own declaration form, we might design the host language's declaration to take a list of declaration modifiers. Then the security and no-zero-division extensions could both add their own modifiers and be used together for the same variable.

Another consideration is in how general we write relations to be. In a non-extensible language, we know exactly how we plan to use a relation and, no other uses being added later, we can write the relation to be specific to that use case. However, in an extensible setting, it might be easier to split one specific relation into a couple of more general ones. For example, rather than write a relation to walk down a list-like abstract syntax tree, gathering information at each node and then carrying out an action over it at the end, it is better to write it as two separate relations. One relation can walk down the abstract syntax tree, gathering information, which is then passed to another relation that carries out the final action. This allows the gathering relation to be reused if an extension wants to repurpose it. In addition to making relations reusable, this can also make it easier to prove properties about it, as the properties can be split into two simpler parts.

Some considerations like these affected the development of AbleC [17], an extensible specification of the C programming language. Its definition was explicitly given certain *extension points*, syntactic and semantic additions to what one might expect of a specification of the C programming language, enabling extensions to add constructs and semantics

they could not add without the extension points. The example of the security and no-zero-division extensions above is similar to type qualifiers in AbleC. The host language is defined so extensions can declare new qualifiers, such as for non-null pointers, and add them to declarations using the host language's syntax. It is also defined to aid extensions in propagating and checking the conditions of qualifiers are satisfied. AbleC also defines the semantics of its operators generally, allowing for overloading by extensions when they add new types. Choices like these can make it easier for extension authors to introduce new and interesting constructs to a language.

7.5.2 Declaring Metatheoretic Properties

Crafting a set of metatheoretic properties for an extensible language module is a balancing act. On one hand, extensions should have freedom to introduce new and interesting semantics. On the other hand, extensions should have access to properties constraining behavior so they can prove interesting new properties. A third consideration, not on the same continuum between semantic freedom and proving ability but also affected by choices there, is in the ease of writing the semantics for extensions. As we saw with V_E in Section 7.4, the choices made by a module can also make new semantic definitions more or less intuitive. The particular choices a language developer makes will depend on what his focus will be for the library and its purpose, whether he wants to enable lots of new syntax, lots of new properties, or make it easy for others to write extensions. We give here some considerations in this balancing act and our experience with them.

Once we have decided our language should have a certain property, we need to decide exactly how to state it. In non-extensible settings, we generally figure out exactly what premises we need to prove our conclusion and give only those, as anything more limits the property's applicability unnecessarily. Making the set of premises minimal, and those premises as general as possible, seems like it is good for extension freedom as well. However, this is not necessarily the case. By minimizing and generalizing the property statement, we might be cutting off some extensions that need extra information, and more specific information, to prove properties. Some extensions may require more specific information

about what is expected to be true when proving a property than is needed in the host language. If there is no reason a more specific version of a premise should not be true, it may be better to require the more specific version. This must, of course, be balanced against the possibility an extension *will* want to break the specific version of the premise. Which way to err, on the general or specific side, may be dependent on the language being defined.

An example of premises in Extensibella we might not need when a property is introduced but might need later is *is* relations as discussed in Section 7.1. While the module introducing a property might not need to use the structure, it can be helpful in extensions. Our list extension to V_L in the previous section projects $cons(e_1, e_2)$ to $eq(e_1, e_2)$, which the projection constraints require to evaluate whenever the original expression evaluates. This then requires knowing that the values v_1 and v_2, to which e_1 and e_2 evaluate, are equal or not. The decidability of equality for values was a property of the host language upon which we implicitly relied. Because the projection constraint's statement in Extensibella includes the *is* relation for the original expression, allowing us to get the *is* relation for the values as well, we are able to show this projection evaluates, something we would not be able to show otherwise.

Another set of considerations are those for what properties to include. Proving properties with imported key relations that don't also involve newly-introduced relations is generally quite difficult; they need to be implied directly by other properties and projection constraints. Thus it is important for a module designer to try to provide *all* properties that are expected to be true about the module's relations and that might be useful for extensions to use as lemmas in their own properties. This also must be balanced against extension freedom, but properties stated generally enough are more often going to be helpful to extension writers trying to prove properties about imported relations than they are going to be harmful to the freedom of extension writers adding to the language.

A particular example of this is introducing the $addP(R)$ property for relations extensions may want to use. We have found two classes of relations extensions often want to use as key relations. The first is evaluation relations. Since extensions commonly introduce

new static analyses intended to indicate something about dynamic behavior, extension-introduced properties often include evaluation as a premise. As evaluation usually does not follow the structure of terms as static analyses do, we often want to use it as the key relation, requiring the $addP(R)$ property for it. The second set, perhaps unexpectedly, are *is* relations. Sometimes we want to show a relation will hold with some arguments for any term, for which we want to carry out induction on the structure of the term. This requires using an *is* relation as the key relation.

Once a set of properties for a module has been decided, they must be given an order. As discussed in Chapter 5, this must maintain the composed order for any imported modules, but new properties may be added anywhere in the order. This order should reflect likely proof dependencies, not only for properties needed as lemmas in the context of the current module, but also considering what properties *might* be useful as lemmas in extended contexts. This is particularly important for projection constraints, as they do not have any projecting syntax creating proof cases when they are introduced to show the module designer what lemmas might be needed. In deciding where to order $addP(R)$ properties, one must also consider when extensions might want to use them to permit a relation's use as a new property's key relation. A good rule of thumb is that the more basic a property seems, the earlier in an order it should go, as it will not require many other properties as lemmas for its proof and it might be needed to prove other properties.

All the considerations we have discussed depend on anticipating what extensions might need. In our experience, this anticipation is the most common source of difficulty in using our reasoning framework as opposed to writing proofs of properties in a monolithic setting. The restrictions from Definitions 4.1 and 4.6 do not generally make proofs more difficult to write, as the nature of metatheory proofs is generally to obey them anyway. Generic proofs are often not difficult to write, usually being easier than at least a few of the proofs for the known cases. In contrast, we have sometimes found while writing examples that our host language had not sufficiently anticipated extension needs; the considerations in this section come from times where the host language was insufficient in its properties, either the ones it had chosen or how it stated them.

To alleviate some of the problem, a module designer might try writing her own extension before declaring her work done. A simple extension adding some new syntax and only properties necessary as lemmas for existing properties does not take too much work to write, but can identify problem points that are not obvious otherwise. For example, consider the unified canonical forms lemma from Section 7.2. Knowing the language includes the progress property, it is clear extensions will need to introduce canonical form lemmas if they introduce new values and syntax operating on those values. However, until we try to prove one for the pair extension it is not clear there will be any problem in doing so, and thus that we need a host-language solution. Trying out the extension-writing experience can also potentially catch problems with property orders and premises, and thus can be considered a good service to provide to other contributors to a language library.

While we have written several example language libraries, much more exploration of our ideas is needed. Three particular topics for further exploration stand out. The first is the applicability of our framework to more extensions that are primarily semantic in nature, adding new semantic relations and properties about them. Next, there is still much space to explore in the trade-offs between extension freedom and the ability to prove properties. We explored some options for definitions of evaluation and projection constraints about evaluation in the previous section. However, we mostly left other semantic relations, such as typing, unexamined. It could be there are interesting and useful projection constraints beyond those for evaluation. Finally, the languages we have written have generally not been realistic. The languages in the previous section, while larger than the others, are still smaller than general programming languages. Specifying and extending realistic programming languages, such as C, could provide more insight into our method's usefulness.

Chapter 8

Related Work

In this thesis we have presented a completely modular approach to developing the metatheory of extensible languages. Any language module may introduce a new metatheoretic property, with the proof for each property spread across modules. A composed language is then guaranteed to have all the properties introduced by the modules included in it based on these modular proofs. In this chapter, we consider related work on modularity in extensible languages, both for specifications of syntax and semantics and for metatheory.

We first consider modular checks for the syntax and semantics of extensible languages that guarantee desirable properties of compositions in Section 8.1. Just as our framework permits different language modules to introduce new metatheoretic properties and write modular proofs guaranteeing they will hold for any composed language, the prior works we consider allow introducing portions of language syntax and semantics in different modules, with guarantees desired properties hold of any composition.

We then consider prior approaches to developing the metatheory of extensible languages in Section 8.2. These works have two primary drawbacks. First, some approaches require *glue proofs*, changes or additions to modular proofs, to complete proof compositions, making them not completely modular. Second, some approaches provide automatic composition of modular proofs, guaranteeing properties will hold for any composition, but do not permit arbitrary modules to introduce new properties, instead requiring all properties to be introduced in a way where all modules will know them. Additionally, we consider another framework that has other severe limitations making it unsuitable for fulfilling our vision of any module introducing new properties and guaranteeing they hold for any composition.

8.1 Modular Guarantees for Language Specifications

Because our goals are to allow non-experts to build language compositions and to do so with the benefits of metatheory for their composed languages, we are also interested in guarantees that language compositions are well-formed in other ways based on modular checks. Such guarantees help fulfill our vision of programmers, not language experts, being able to choose sets of modules from a language and use the language resulting from their composition. Our guarantee that any composed language will have certain metatheoretic properties is useless for such programmers unless the composed language is also guaranteed to have well-formed concrete syntax and the programs they write can be parsed, and that it has well-formed semantics so it can be executed or interpreted. We examine here three such guarantees based on modular checks.

The first guarantee is relatively simple. Consider typing for language specifications. In Section 2.2, we mentioned a requirement that the rules introduced by language specification modules must be consistent with the typing constraints imposed by the specifications of constructors and relations. A modular check that each module's specification is well-typed guarantees a composed language then is also well-typed. Similarly, in the Silver attribute grammar system [38], language modules are checked for type errors in a modular fashion, which then guarantees composed languages are well-typed. It is easy to see that modular typing ensures compositions will be well-typed. A well-typed specification is one where each individual rule, or, for attribute grammars, each individual equation, is well-typed. Thus taking the union of individual sets where each element is well-typed is also well-typed.

In contrast to the inherent modularity of typing, consider grammars as specifications of concrete syntax. Grammars consist of sets of terminals, nonterminals, and productions. The composition of grammars from independent modules can be implemented as a union of the corresponding sets from each module, as with most parts of our semantic specifications. However, the property we might desire of this composition is more complex to maintain in composition than typing. We generally want a grammar for a programming language to have a deterministic parser, guaranteeing each program in the grammar's language can

be parsed in a unique way. Some systems for extensible languages with concrete syntax specifications, such as SugarJ [9], Sugar* [10], and Spoofax [20], use generalized parsing. This allows ambiguous composed grammars to be parsed, and thus the same program may have multiple parses, and possibly multiple meanings, only one of which the programmer writing it intended. In contrast, modular determinism analysis (MDA) [36], implemented by the Copper parser generator [25], provides a modular check of grammar additions from extension modules relative to a host language's grammar to ensure their compositionality. A set of extensions that all individually pass this modular check are guaranteed to compose such that a specific combined approach to scanning and parsing produces a deterministic parser for the composed language. Requiring extensions to pass the MDA check may rule out some syntax an extension developer wants to introduce as it cannot pass the check even though particular compositions might be fine. However, it also eliminates the possibility of creating a composed language that cannot be deterministically parsed.

Another guarantee based on a modular check, this one for language semantics, comes from the setting of attribute grammars. When an attribute grammar is animated to be used as a compiler or interpreter, a missing attribute equation means it will crash; this is analogous to encountering a missing pattern in pattern matching in a functional language. In the JastAdd [13] attribute grammar system, used for writing extensible languages, a check for missing equations is carried out when a composed language is built. If equations are missing at that time, the language composer must write glue code to provide them, a non-modular approach to language composition. To have a completely modular approach to composition, this check must be turned into one individual modules can carry out. Modular well-definedness analysis (MWDA) [18], implemented in the Silver system [38], creates such a modular check. MWDA uses a flow analysis to determine on which attributes a particular one may depend, with this analysis remaining valid even when more constructs are added to the language. The flow analysis determines which other attributes an attribute access may need, and thus, across all possible attribute accesses, which equations are necessary. If all modules in a composition pass this check, the composed language is guaranteed to have all the equations it can need, and cannot crash due to a missing equation.

Our work applies this same approach of modular checks giving compositional guarantees to metatheory. As in MDA, where any module may introduce new syntax, and MWDA, where any module may introduce new semantics, our framework permits any module to introduce new metatheoretic properties. As typing, MDA, and MWDA have modular checks that modules are well-formed in desirable ways, so our reasoning framework has a modular check in the form of modular proofs. We then have a guarantee of all metatheoretic properties holding, just as typing has a guarantee of composed well-typedness, MDA has a guarantee of determinism in parsing, and MWDA has a guarantee of no missing equations.

8.2 Metatheory Frameworks for Language Extension

In contrast to our approach to metatheory, which builds metatheory modularly and guarantees it holds for any composed language, most existing frameworks for developing the metatheory of extensible languages either do not permit arbitrary language modules to introduce new properties, or they do not provide a guarantee properties will hold for composed languages. In this section we examine prior approaches to metatheory for extensible languages, breaking them down by whether their primary limitation relative to our approach is limiting the introduction of new properties or requiring glue code for composition. We also consider an approach to developing metatheory for extensible languages, called non-interference, that superficially shares the same characteristics as our framework but has other serious limitations.

8.2.1 Frameworks Limiting Properties

There are several frameworks for proving properties about extensible languages that limit how properties may be introduced, but guarantee the properties they do have hold for composed languages. These operate on varying models of language extension.

SoundX

The first framework we consider is SoundX [23]. SoundX assumes a language extensibility framework where a host language introduces abstract syntax and defines typing relations, with extensions adding new syntax, defining the typing relations for that syntax, and defining translations of the new syntax into the host language, similar to our projections. Typing is checked for programs using the extended syntax, then translated to the host language for any further analysis, making the extensions little more than syntactic sugar. SoundX proves one property, that programs well-typed with the extension typing rules will desugar via their translations into well-typed programs in the host language alone. Desugared expressions having the same type as the original expressions, but desugared (*e.g.*, for the pair extension from Section 7.2, an expression of type $pairTy(intTy)$ desugars to an expression of the arrow type to which $pairTy(intTy)$ projects). SoundX proves this automatically for each rule an extension introduces, creating an equivalent typing derivation for the full translation using only rules introduced by the host language. This property permits the host language designer to prove properties he desires, such as type soundness, outside of the SoundX framework for the host language alone and know they will hold for any composed language, since only fully-desugared programs are evaluated.

The drawback of this approach is that it is limited to this one property. It cannot be used to prove anything beyond a direct correspondence between sugar and desugared derivations of typing; any other properties must be proven separately for the host language alone. Such a direct correspondence may also be too strict to permit some interesting extensions, even if a language designer is not concerned about extending other relations or proving further properties. Our list extension from Chapter 2 does not conform to this, as the projections do not have related types, nor do our list extensions from Section 7.4 where lists project to records. The type $listTy(intTy)$ projects to a record type with a *head* field of type $intTy$, a *tail* field of type $listTy(intTy)$, and a *null* field of type $boolTy$. The projection of $cons(e_1, e_2)$ has all these fields, but the projection of *nil* only has the *null* field, breaking SoundX's property. This suggests SoundX's approach might be limited in what

new datatypes extensions can add, particularly if the host language is relatively simple. In contrast, our approach permits language designers to choose what they expect to be true of projections, giving extensions more freedom. This allows extension-introduced syntax to be more than syntactic sugar, as it must be with SoundX.

Extending Inductive Types

Another work, by Boite [3], allows more freedom than SoundX as properties are introduced by the initiator of the language, not by the reasoning framework itself. It looks at reusing proofs when extending inductive types with additional constructors and modifying existing relations by adding new rules and new parameters. All reasoning is carried out in Coq [1]. The language framework underlying this work assumes a host language module introduces a set of syntax categories and relations, with extension modules adding new constructors of those categories and new rules for the relations. Additionally, relations can be modified by adding new parameters, such as an extension adding a new parameter for a variable store to an existing evaluation relation without one, with existing rules modified automatically to include new parameters. Because of these modifications of relations, extensions can only be added linearly, building on each other in sequence; independent extensions that are then composed are not allowed. The reasoning framework allows the host language module to introduce a set of properties for the language, which the host language designer proves for the host language without any restrictions on proofs. Each successive extension is then required only to add new proofs for cases arising from case analyses for the new rules it adds, with the cases from the modules on which it builds reusing the proofs already written for them. Because extension is linear, it needs no restrictions on how proofs are written, as opposed to the restrictions we must place on proofs. Any case analysis is valid, as extensions adding new cases even to nested, non-top-level case analyses will know these new cases need to be proven.

While the soundness of proofs is guaranteed, it appears the reasoning framework of this work is underpowered relative to the language framework about which it reasons. Extensions may add new parameters to relations, but they cannot introduce new properties about those

parameters, nor can they add well-formedness conditions about them as premises for existing properties. The language framework is also quite restrictive compared to ours. In addition to not allowing extensions to introduce new relations, the limitation to linear extension means users of such languages cannot pick and choose extensions; a user may select the extensions he wants, but must then use all extensions in the extension sequence up to the last one he wants, including any intervening ones he does not want. Linear extension also means extension developers must coordinate to decide whose extension will be next in line; they cannot independently write their own features.

Meta-Theory à la Carte

Our final example of an existing framework with sound composition is meta-theory à la carte (MTC) [7]. MTC reasons about languages written using what we call the complementary components framework, where independent component modules build on a set of shared declarations of syntax categories and relations. Each module adds new constructors of the shared categories and new rules defining the shared relations. MTC also includes metatheoretic properties as part of the shared declarations, with each module writing proofs of the properties for the cases it introduces. These proofs also have restrictions on case analysis, similar to those imposed by our Definitions 4.1 and 4.6, to ensure composed proofs are complete. These proofs are written in Coq using a particular, complex encoding of languages and proofs. This encoding then allows all the component modules and proofs to be composed, forming a single language specification and proofs of properties about it. Unfortunately, this encoding is so complex it can be difficult to ascertain whether the language written, and each property, is the one intended; the authors note this difficulty in their paper. This is in contrast to our approach, where the encoding into $\mathcal{G}$ is very direct and the correspondence between our language specification and the one about which we reason is clear.

The benefit of this encoding of languages and proofs is that it builds the proof restrictions and composition directly into Coq's logic. In contrast, our proof restrictions need

to be enforced on top of $\mathcal{G}$ as an extra check for modular validity, and our proof composition requires pulling apart $\mathcal{G}$ proofs and recombining their modified sub-proofs to create composed ones.

Shared Limitations of These Frameworks

All of these reasoning frameworks are lacking relative to ours because modules other than a host language, if that, cannot introduce new properties, nor can their underlying language frameworks allow extensions to introduce new relations. This means that at best they can support extensions extending the existing relations in interesting ways; SoundX cannot even support this, as its extensions are syntactic sugar and their semantics must exactly match their translations. Thus these frameworks cannot support semantic extensions, such as our security and optimization extensions from Chapter 2, meaning their modularity is limited to extensions that are basically syntactic in nature.

In practice, this type of limitation, where modules cannot introduce new properties, is often too strong to be useful. Even proving existing properties, such as ones that might be introduced by the host language in Boite's framework or that might be part of the shared declarations of MTC, often requires using others as lemmas. Some such lemmas may need to be introduced by extensions to support their new definitions, properties that cannot be stated by the host language because the specific syntax referenced in the property is not known at that time. For this reason, the implementation of the MTC framework bows to practicality and allows component modules to introduce new properties, but compositions including such properties require glue proofs to complete them. Shared properties are still proven without requiring glue proofs, once the lemmas they need have been glued into the proof development.

8.2.2 Frameworks Requiring Glue Proofs

We turn now to discussing reasoning frameworks where modular reasoning does not guarantee properties will hold for any composed language, and thus proof composition might require glue proofs to make separate pieces work together. These works provide ways of

reusing proofs of properties for modules in different language extensibility frameworks, reducing the work needed for proving properties for a composition, but not eliminating it. The frameworks in the previous section, being completely modular, allow programmers who are not experts in language design or logic to choose what extensions to use and know the properties of the full language. The frameworks in this section instead require experts to create compositions of sets of extensions, as the composed proofs might require extra work to finish. Additionally, properties with semi-modular proofs, as these frameworks have, may not hold for every composition due to interactions between the different features included in compositions. No guarantee is given until an expert creates and certifies a composition's proofs.

Proof Weaving

An early such work, Proof Weaving [30], reasons about languages written using the complementary components framework, similar to MTC. Each component writes its proof in Coq for each property, whether that be a property that is part of the shared set of declarations or specific to the component, with no restrictions. The proofs for each component in a composition are then combined (or *woven*) to create a proof for the property in the full language. However, this proof may not be complete. Because there are no restrictions on proofs, the composed proof may have holes, such as from a case analysis where only a portion of the cases were known at the time the proof was written. The implementation tries to fill these holes, but in general the human composer of the proofs must write glue proofs to fill the holes instead. Overall, Proof Weaving is less a principled framework for developing metatheory and more a computer program for manipulating Coq developments in ways that may be useful, but are not guaranteed to be successful.

Modular Monadic Metatheory Library

The next work we consider is an extension to MTC, the modular monadic metatheory library (M^3TL) [5]. It reasons about languages using a modification of the complementary components framework where each component module can use monads to add effects to

shared relations, such as exceptions, variable stores, or references. The reasoning framework assumes a set of shared property declarations, but each component module declares and proves its own version of each property to fit the effects it adds (*e.g.*, a proof of type preservation for a component module adding a variable store needs to add a premise that the variable stores for typing and evaluation correspond). These changes must meet a formal requirement ensuring the set of effects is abstract enough to permit other components also to introduce new effects. The M^{3}TL framework reuses these modular proofs, along with glue code for combining the included monads and glue proofs for combining the modified property statements and their proofs, to create composed proofs of properties. As with the underlying MTC framework, the encodings of languages, properties, and proofs for M^{3}TL are quite complex. The inclusion of effects makes the encodings significantly more complex than those found in MTC.

M^{3}TL provides a feature our reasoning framework does not, that of adding effects to a language and reasoning about them. Our language extensibility framework can support the addition of effects by extensions via a clever use of existing relation arguments, if the host language plans for it.[1] However, it appears our reasoning framework cannot reason about them as M^{3}TL does because we cannot modify existing property statements to add extra premises for the well-formedness of new effects, nor change their conclusions to reflect the inclusion of new effects.

Product Lines of Theorems

Another work, by Delaware *et al.* [6], treats language extension as product lines, with each extension being a feature optionally included in the language. The idea underlying this approach is treating a language as a software product with different versions. A software product might have a basic version with few features, a pro version with a full set of features, and in-between versions that include all the features of the basic version plus some subset of those in the pro version. The higher-featured versions of the software product use

[1] Sterling code for a language modeling the approach to language extension of modular structural operational semantics [29], which also permits adding effects to languages, can be found online at `https://mmel.cs.umn.edu/sterling/examples/modelingMSOS/description.html`.

the same code as the more basic ones, all developed by the same team of developers, but with certain program modules added to implement the added features, where the modules may also modify the existing code in some ways. Treating language extension as product lines also assumes one team develops all the language features, with different versions of the language resulting from including different extension feature modules relative to the host language. The host language introduces variation points extensions may use to modify existing definitions, adding new arguments to constructors and new parameters to relations, as well as adding new premises to rules. This is similar to the approach to extension taken by M^3TL. Due to these changes to the shared definitions, glue code might be needed to create composed languages including different features. Because of the assumption that one team develops all the features, and thus knows all the possible combinations, the team can also develop all the glue that might be needed.

The host language also introduces properties for the language, treating the property statements and their proofs as a product line as well. As with language definitions, the property statements also include variation points, allowing language features to modify them to fit the new features they may add. These variation points have restrictions on what extensions may add so different property versions can be composed. Each module writes a proof in Coq for the cases it introduces, with appropriate restrictions on the proofs so they may be composed. A metatheory composition requires glue proofs combining the modified property statements and filling in the changes to other proofs to fit the property statements modified for full sets of effects, creating full property statements combining all variations and valid proofs for the full properties.

Because of the explicit assumption that product-line extensible languages are developed by one team, the glue code and glue proofs required for composition can be developed by that same team. This allows non-experts to choose the features they want from the set provided by the language developers without concerns about compositions, as all possible compositions have been certified by the language experts who wrote them. The drawback of this approach relative to our vision is that, while programmers are free to choose any set of language features they want, outside developers are not free to add new feature modules

to the language. Only the central team of language developers can add modules, as they need to certify all possible compositions for soundness.

FPOP

Our last framework requiring glue proofs is FPOP [14]. Its language extensibility framework allows extensions to build on a host language and other extensions to it. In doing so, an extension module inherits their definitions, adding its own definitions to them. It may also modify imported definitions via glue code, changing the definitions given by other modules, in addition to introducing its own new relations and defining them. Properties and their proofs are treated similarly to semantic definitions. Extensions may introduce their own properties, and must prove properties inherited from other modules as well. For existing properties, an extension may either extend the existing proof, writing proofs only for the new cases it introduces, or it may rewrite the entire proof from scratch. The latter can be useful if the original proof's structure does not work with the additions an extension makes to the language.

Rewriting proofs from scratch is enabled by FPOP's lack of a concept of automatic composition. It is freer than Boite's work from the previous section in that linear extension is not a requirement; two modules may both build on the same module. However, combining independent modules requires a language developer to write a new module inheriting from both. In general, a combining module may need to rewrite the proofs of the shared properties of independent modules it includes from scratch, as they may not share the same structure. This metatheory framework places a heavy burden on extension writers in comparison with our approach, as writing extensions in FPOP may require rewriting the entirety of proofs for existing properties, whereas our approach only requires an extension writer to add new cases to existing properties' proofs.

8.2.3 The Non-Interference Framework

The final framework we consider [19] introduces a notion it calls *interference*, undesirable interactions between extensions that invalidate properties, and defines *non-interference* of

extensions as a way to ensure desired properties hold for composed languages. Superficially, this framework appears to support many of the same things as ours. It does not limit properties as we saw in Section 8.2.1, as any language module may introduce new properties. Property proofs also do not require glue proofs for composition as in the frameworks in Section 8.2.2. It also assumes Silver's model of extensibility [38], of which ours is a generalization. Despite these similarities, there are several limitations of this framework relative to ours.

The non-interference framework requires attribute equations for a term and its projection always to evaluate to essentially-equal values, where the values are either the same or one projects to the other; extensions obeying this restriction are said to be non-interfering. This mirrors SoundX's requirement for typing, requiring that a term and its projection must have types that are equal or projections, but extends it to the definitions of all semantics. Properties are proven for the host language, and they are proven to hold when two terms have semantic attributes with essentially-equal values. Then the composed proof of a property can apply the given proof to the host language and copy the property's truth for terms built by extension-introduced syntax constructors from their projections because all a term's semantic attributes must have values that are essentially equal to its projection's values. Copying the property from the projection like this is similar to our generic proofs, but with a more simplistic, one-approach-fits-all generic proof. The proof work required by the non-interference framework then is three-pronged: (1) extensions prove their attribute equations for new syntax evaluate to values that are essentially equal to the corresponding attributes for the syntax's projection, (2) modules prove their new properties for the host language, and (3) modules prove their new properties hold for any terms with essentially-equal attribute values.

One difficulty in using this framework is that non-interference places very strict requirements on extensions. Our list extension from Chapter 2 is not non-interfering, as projections such as projecting $cons(e_1, e_2)$ to $eq(e_1, e_2)$ do not maintain the same semantics. Consider also the *secdecl* construct from the security extension from Chapter 2. The updated security context for *secdecl* can differ from that for the *decl* to which it projects, as the *secdecl* can

be used to declare variables as *private*, whereas *decl* always declares variables as *public*. Without being able to declare variables as private, as this requirement does not permit, the security analysis can be introduced and proven to ensure private information is not leaked, but it cannot be useful. The requirement for each of a term's attribute values to be related to the corresponding attribute value of its projection is like a very strict projection constraint. By making the requirements for projections be specified by the language designer rather than a characteristic of the framework itself, we allow each language library to choose requirements that work well for it.

Another limitation of this framework is that its property proofs are only applicable to fully-decorated trees, meaning terms for which all semantic analyses have been derived. This immediately rules out some ill-behaved trees, such as those without types or that do not evaluate. While it might appear the modular well-definedness analysis would ensure such trees do not exist, MWDA only considers the *existence* of attribute equations; it cannot consider whether the equations will produce values or not. In particular, the non-interference framework rules out considering trees where the computation of some attribute value is non-terminating. Then a property about the existence of a derivation of a semantic relation is vacuously true in this framework, as terms not having such a derivation are excluded from consideration. This can be mitigated somewhat by an existing analysis that ensures computations in an attribute grammar terminate [21]. However, this analysis is naturally conservative, and encodings of many languages, such as statement evaluation in our example language from Chapter 2, into an attribute grammar cannot pass it. Such a determination would require the analysis to determine any program written in the language would terminate.

Finally, despite allowing any module to introduce a new property, not just the host language, this framework does not consider the use of properties as lemmas. The author states he believes circularity in proofs of different lemmas cannot be introduced as long as each module's proofs are fine individually [16]. However, as no module explicitly writes the proofs for copying a property from a term's projection, it is not clear circular dependencies cannot be introduced in the composition by this process.

Chapter 9

Conclusion

This thesis has presented a reasoning framework for developing the metatheory of extensible languages in a modular fashion. In our framework, we assume extensible languages are developed as composable modules in language libraries, with the modules introducing language features. A common structure for language libraries is to have a module defining a host language, with other modules being extensions to the host language. Extension modules extend the definitions given by the host language, adding new syntax and new semantics to the language. Extensions may be developed independently of one another, so the syntax and semantics they introduce are not known by other modules. Language composition combines the definitions given by all modules, creating a full language with all features introduced by all included modules. This composition includes extending the definitions of new semantic relations introduced by one module to the new constructs introduced by an independent module, often using *projections* that translate away extension-introduced constructs.

Our reasoning framework permits language modules to introduce their own new properties, with a guarantee each property will hold for any composition of well-formed modules. The proof that a property will hold for any composed language is distributed across the set of modules knowing the property as *modular proofs*, each providing a portion of the proof work necessary for a composed language. Each modular proof is written in the context of a single module's knowledge of the language, proving the property for the portion of the language it knows. However, not all parts of a language are known by a module that also knows the property. When an extension module introduces a property, other extensions

that are independent of it but which may be included in a composed language cannot contribute to the proof, although they may contribute to the definitions for which the property specifies relationships. Thus we also identify a way for modular proofs to use *generic* reasoning, allowing a module introducing a property to prove it will hold for portions of the language introduced by independent modules not knowing the property. This generic reasoning is made possible by using other properties as lemmas and knowing how definitions are extended to constructs from other extensions. In particular, properties we identify as *projection constraints*, which specify how the semantics of a construct and its projection must be related, figure prominently in such generic reasoning. The modular proofs written by the modules included in a language composition may be composed to form the full proof necessary for each property for a composed language, similar to how the modules in the language specification are composed to form a full language.

The use of generic reasoning makes it possible for *any* module to introduce new metatheoretic properties. Without generic reasoning to extend proofs to constructs that are not modularly known with the property, only the host language would be able to introduce properties. Because the host language is known to all modules in a language library, properties introduced by it are also known, and thus modular proofs of them can be written by all modules. As extensions are not known to all modules, their properties are not shared by all. If extensions could not introduce new properties, they could not usefully introduce new semantic analyses, such as the security check for information flow mentioned in Chapter 1 and expanded upon in Chapter 2. Without properties guaranteeing the accuracy of such checks, they would not be useful. Then extensions would be limited to adding new syntax and extending existing semantics to them only, in contrast to the full participation in defining the language by also introducing new semantic analyses that we have presented.

In addition to developing our reasoning framework, we have written an implementation of it. To support this, we have also written an implementation of the language extensibility framework on which it relies. Together, these allow language developers to write specifications of language modules and write modular proofs of their metatheoretic properties. Using these implementations, we have created several example language libraries with

metatheoretic properties and modular proofs of them. Overall, these have suggested our reasoning framework is practical for the development of modular metatheory, and that the limitations it imposes on proofs are not too restrictive. These examples have also allowed us to examine the effects different choices of projection constraints have on what extension modules may introduce. Because the generic reasoning needed for proving properties introduced by extensions relies so heavily on projection constraints, the choices of projection constraints affect what properties extensions can prove. Additionally, because extensions must prove the projection constraints hold for their new syntax, they affect what syntax extensions may introduce, and how they define the semantics of that syntax. The ability of extensions to prove interesting properties and the freedom of extensions to introduce interesting syntax is then in tension, as more restrictive projection constraints help prove properties but limit syntax, and less restrictive projection constraints do the opposite. In investigating these trade-offs, we have found there is no perfect balance between the two. The choice must be made by the language designers, deciding what is appropriate for the language they are writing, taking from our experiences advice on what may be appropriate.

Our reasoning framework provides a means for realizing the vision of extensible languages we laid out in the introduction relative to metatheory. In this vision, programmers who are not experts in developing programming languages can choose the features they want in a programming language from the modules in a language library, with automatic composition of the modules to form a full language with all the features from the modules they selected. The reasoning framework in this thesis allows them also to compose the metatheory of the language automatically, guaranteeing the full language has all the metatheoretic properties from the modules they selected. However, there is more work that can be done.

One aspect of this work is writing more example languages. As noted at the end of Chapter 7, there is more room for trying out different possible projection constraints, determining how different choices affect extensions and when they are useful, and for trying out more extensions to more types of languages than we have been able to write. We also noted at the end of Chapter 5 that determining what ordering choices are useful in different

situations requires developing more languages, particularly ones with extensions building on other extensions. Finally, even our largest languages are still toy languages. Our framework needs to be tested for developing the metatheory of realistic languages as well.

In addition to writing more examples, we can improve the experience of module authors developing modular metatheory and the experience of programmers relying on it. We expand on these ideas in the rest of this chapter, presenting possibilities for future work along these lines.

9.1 Higher-Order Abstract Syntax for Reasoning

Higher-order abstract syntax [28, 34] is a method for encoding binding structures in object-level abstract syntax by using binding structures in the meta-language in which it is encoded. This approach can improve ease of specification and, especially, ease of reasoning about programming languages [40, 41]. Using higher-order abstract syntax, binding structures for constructs such as variable names can be represented using meta-level bindings. This eliminates the need for handling bindings explicitly, such as the approach using typing and evaluation contexts we had in our example language in Chapter 2. Reasoning about the object language can take advantage of metatheoretic properties about such bindings rather than proving them specifically for the object language. For example, weakening for typing (*i.e.*, adding new type bindings to a context does not change the type of an expression) when using higher-order abstract syntax can rely on weakening as a metatheoretic property of the meta-level bindings. This eliminates the work of proving weakening specifically for typing; the weakening for typing follows directly from weakening for the meta-level bindings. Because we have limited ourselves to a first-order setting, we cannot use higher-order abstract syntax, and must handle binding structures explicitly in our language encodings and proofs.

To use higher-order abstract syntax in our work, we would need to leave the first-order setting. Fortunately, our chosen logic $\mathcal{G}$ [11] already supports reasoning about higher-order abstract syntax, so most of our work on the reasoning framework would still apply. The main

apparent technical difficulty is in the relationship between unification in the limited context of the language of a single module and the larger context of a composed language, and between unification before and after term replacement eliminating generic constructors. In our lifting of generic proofs to new constructs in the context of composed languages, replacing generic constructors with terms built by new constructors relies on knowing unification with a formula before term replacement implies unification with a related formula after term replacement, and that the most general unifiers from both unifications are related by term replacement and substitution as well. This is used in the cases for proof rules using language rules in the reasoning, either through case analysis or applying a language rule to a sequent's conclusion. The difficulties in moving to a higher-order setting that can support higher-order abstract syntax would likely primarily arise in proving Theorem 3.4 and the theorems in Appendix B due to the use of higher-order unification.

A less-technical aspect of introducing higher-order abstract syntax would be considerations on whether its use affects extensibility. We saw in Chapter 7 that some choices we make in the syntax of extensible languages affect what extensions can introduce. Once the theory allowing higher-order abstract syntax in an extended version of our reasoning framework is completed, an exploration of the effect of using it on extensibility of languages can be undertaken. In particular, this can focus on whether defining extensible higher-order abstract syntax restricts what extensions may introduce and whether it has an effect on the ease of introducing properties requiring generic reasoning.

9.2 Property-Based Testing for Modular Metatheory

Proving language properties is quite time-consuming, even when the proofs are not particularly complex. This is a time investment many language designers are not willing to make. However, even when language designers do not intend to expend the time and effort required to verify properties hold, it can be still be useful to *identify* the properties expected to hold for a language. Knowing what properties a language is expected to have can be useful for programmers using it for non-critical applications. If a language is supposed to

have the property of type preservation, and thus well-typed programs should not encounter ill-typed values, a programmer can choose to write a program assuming all values will have the correct types and not include checks that values do have the expected forms.

For situations where full verification is not necessary, we might be able to get a reasonable level of assurance from property-based testing [4]. In general property-based testing, a property a function is expected to have is identified, then randomized inputs are generated for the function, checking each output satisfies the expected property. For example, we might check a function f is commutative by generating random inputs a and b and checking that $f(a, b) = f(b, a)$. This is a much lighter-weight approach to checking properties hold as determining the properties desired of a program, or of a language, is relatively easy compared to proving them.

It might be possible to apply property-based testing to the modular development of metatheory. This would allow module designers to introduce the properties they expect to be true and get some modular assurance they hold for composed languages. Using our reasoning framework as a guide to what to test, modules might test the properties we want the composed language to have, like type preservation or the security property, and also projection constraints that help module authors determine if their default rules are sensible for the projections they use. The difficulty, as we saw with writing modular proofs, is that only a portion of the language is known modularly. This suggests a robust testing-based approach to modular metatheory should be able to introduce some sort of unknowns into the testing, both to represent generic proof cases and the possibility that a sub-term will be constructed using a rule introduced by an unknown extension that defines the relation a bit differently than any known rule.

9.3 Safe Relation Orders

While our focus in this work is on specifications of extensible languages and reasoning about them, our vision is for extensible languages to be useful to programmers. Programmers choose the language modules they want to use and compose them, with our work ensuring

their composed languages have the metatheoretic properties the designers of all the modules wanted them to have.

One part of enabling programmers to use extensible languages in their day-to-day programming tasks is turning language specifications into usable languages. This requires creating compilers or interpreters based on those specifications, a problem is beyond the scope of our work, and generally unrelated to our primary concern, metatheory. However, there is one aspect of this related to metatheory, which we consider here, and that is guiding how relations are ordered to be applied in compilers and interpreters.

Consider using our example language from Chapter 2 to execute programs. If we have a composed language including all the modules we have discussed, we might check whether a program is typable using the host language's typing relation and whether it is secure using the security extension's *secure* relation. Once it has passed both checks, we might optimize the program, then execute the optimized program. We modularly proved the original program will not leak sensitive information when executed, but the original program is not what is being executed in this scenario. Is the execution still leak-free? If the optimization extension modularly proves a property that it does not change evaluation results, this is true. However, if we include a different program transformation that is not verified to do so, this may not be the case.

Our approach to modular metatheory has guaranteed composed languages have desirable properties. However, as we see in this example, it has not, and indeed cannot, guarantee these properties are applicable to what happens in a use of the language. Future work must address this problem, finding a way to ensure the properties a language has apply to the way it is used. This is an inherently compositional problem, as it does not arise until a composed language is being used, and thus our vision of programmers choosing their own language features further requires a solution to it that does not rely on an expert in languages and logic checking whether an order is safe.

Bibliography

[1] The Coq proof assistant. `http://coq.inria.fr`.

[2] Baelde, D., Chaudhuri, K., Gacek, A., Miller, D., Nadathur, G., Tiu, A., and Wang, Y. Abella: A system for reasoning about relational specifications. *Journal of Formalized Reasoning 7*, 2 (December 2014).

[3] Boite, O. Proof reuse with extended inductive types. In *Theorem Proving in Higher Order Logics* (Berlin, Heidelberg, 2004), K. Slind, A. Bunker, and G. Gopalakrishnan, Eds., Springer Berlin Heidelberg, pp. 50–65.

[4] Claessen, K., and Hughes, J. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2000), ICFP '00, ACM Press, p. 268–279.

[5] Delaware, B. *Feature Modularity in Mechanized Reasoning*. PhD thesis, University of Texas at Austin, Austin, Texas, USA, 2013. `https://repositories.lib.utexas.edu/items/b4d83445-bf35-42ab-81b4-12f6d50e65b9`.

[6] Delaware, B., Cook, W., and Batory, D. Product lines of theorems. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (New York, NY, USA, 2011), OOPSLA, ACM Press, p. 595–608.

[7] Delaware, B., d. S. Oliveira, B. C., and Schrijvers, T. Meta-theory à la carte. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on*

Principles of Programming Languages (New York, NY, USA, 2013), POPL '13, ACM Press, pp. 207–218.

[8] Ekman, T., and Hedin, G. The JastAdd extensible Java compiler. In *Proceedings of the Conference on Object Oriented Programming, Systems, Languages, and Systems (OOPSLA)* (2007), ACM Press, pp. 1–18.

[9] Erdweg, S., Rendel, T., Kastner, C., and Ostermann, K. SugarJ: Library-based syntactic language extensibility. In *Proceedings of the Conference on Object Oriented Programming, Systems, Languages, and Systems (OOPSLA)* (2011), ACM Press, pp. 391–406.

[10] Erdweg, S., and Rieger, F. A framework for extensible languages. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences* (New York, NY, USA, 2013), GPCE '13, ACM Press, p. 3–12.

[11] Gacek, A. *A Framework for Specifying, Prototyping, and Reasoning about Computational Systems*. PhD thesis, University of Minnesota, 2009. `https://arxiv.org/abs/0910.0747`.

[12] Gacek, A., Miller, D., and Nadathur, G. Nominal abstraction. *Information and Computation 209*, 1 (2011), 48–73.

[13] Hedin, G., and Magnusson, E. Jastadd—an aspect-oriented compiler construction system. *Science of Computer Programming 47*, 1 (2003), 37–58.

[14] Jin, E., Amin, N., and Zhang, Y. Extensible metatheory mechanization via family polymorphism. *Proc. ACM Program. Lang. 7*, PLDI (June 2023).

[15] Kaminski, T. *Reliably Composable Language Extensions*. PhD thesis, University of Minnesota, Minneapolis, Minnesota, USA, 2017. `http://hdl.handle.net/11299/188954`.

[16] Kaminski, T. Personal communication, February 2024.

[17] Kaminski, T., Kramer, L., Carlson, T., and Van Wyk, E. Reliable and automatic composition of language extensions to C: The ableC extensible language framework. *Proceedings of the ACM on Programming Languages 1*, OOPSLA (Oct. 2017), 98:1–98:29.

[18] Kaminski, T., and Van Wyk, E. Modular well-definedness analysis for attribute grammars. In *Proceedings of the 5th International Conference on Software Language Engineering (SLE)* (Sept. 2012), vol. 7745 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 352–371.

[19] Kaminski, T., and Van Wyk, E. Ensuring non-interference of composable language extensions. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering (SLE)* (October 2017), ACM Press, pp. 163–174.

[20] Kats, L. C., and Visser, E. The Spoofax language workbench: Rules for declarative specification of languages and IDEs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (New York, NY, USA, 2010), OOPSLA '10, ACM Press, p. 444–463.

[21] Krishnan, L., and Van Wyk, E. Monolithic and modular termination analysis for higher-order attribute grammars. *Science of Computer Programming 96* (December 2014), 511–526.

[22] Leduc, M., Degueule, T., and Combemale, B. Modular language composition for the masses. In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering* (New York, NY, USA, 2018), SLE 2018, ACM Press, p. 47–59.

[23] Lorenzen, F., and Erdweg, S. Sound type-dependent syntactic language extension. *SIGPLAN Not. 51*, 1 (Jan. 2016), 204–216.

[24] Martelli, A., and Montanari, U. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems 4*, 2 (Apr. 1982), 258–282.

[25] MELT group. Copper context-aware scanner and parser generator website. `http://melt.cs.umn.edu/copper`.

[26] Michaelson, D., Nadathur, G., and Van Wyk, E. Modular metatheory for extensible languages webpage. `http://mmel.cs.umn.edu`.

[27] Michaelson, D., Nadathur, G., and Van Wyk, E. A modular approach to metatheoretic reasoning for extensible languages. arXiv manuscript available from `https://arxiv.org/abs/2312.14374`, Dec. 2023.

[28] Miller, D., and Nadathur, G. A logic programming approach to manipulating formulas and programs. In *IEEE Symposium on Logic Programming* (September 1987), S. Haridi, Ed., pp. 379–388.

[29] Mosses, P. D. Modular structural operational semantics. *The Journal of Logic and Algebraic Programming 60-61* (2004), 195–228.

[30] Mulhern, A. Proof weaving. Proceedings of the First Informal ACM SIGPLAN Workshop on Mechanizing Metatheory, September 2006. `https://pages.cs.wisc.edu/~mulhern/Mul2006/Mul2006.pdf`.

[31] Myers, A. C. JFlow: practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1999), POPL '99, ACM Press, p. 228–241.

[32] Myers, A. C., and Liskov, B. A decentralized model for information flow control. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1997), SOSP '97, ACM Press, p. 129–142.

[33] Nystrom, N., Clarkson, M. R., and Myers, A. C. Polyglot: An extensible compiler framework for Java. In *Compiler Construction* (Berlin, Heidelberg, 2003), G. Hedin, Ed., Springer Berlin Heidelberg, pp. 138–152.

[34] Pfenning, F., and Elliott, C. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1988), PLDI '88, ACM Press, p. 199–208.

[35] Schwaab, C., and Siek, J. G. Modular Type-Safety Proofs in Agda. In *Proceedings of the 7th Workshop on Programming Languages Meets Program Verification* (New York, NY, USA, 2013), PLPV '13, ACM Press, pp. 3–12.

[36] Schwerdfeger, A., and Van Wyk, E. Verifiable composition of deterministic grammars. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2009), ACM Press, pp. 199–210.

[37] Tiu, A. *A Logical Framework for Reasoning about Logical Specifications*. PhD thesis, Pennsylvania State University, May 2004. `http://etda.libraries.psu.edu/theses/approved/WorldWideIndex/ETD-479/`.

[38] Van Wyk, E., Bodin, D., Gao, J., and Krishnan, L. Silver: an extensible attribute grammar system. *Science of Computer Programming 75*, 1–2 (January 2010), 39–54.

[39] Van Wyk, E., de Moor, O., Backhouse, K., and Kwiatkowski, P. Forwarding in attribute grammars for modular language design. In *Proceedings of the 11th Conference on Compiler Construction (CC)* (2002), vol. 2304 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 128–142.

[40] Wang, Y. *A Higher-Order Abstract Syntax Approach to the Verified Compilation of Functional Programs*. PhD thesis, University of Minnesota, Dec. 2016. `https://arxiv.org/abs/1702.03363`.

[41] Wang, Y., and Nadathur, G. A higher-order abstract syntax approach to verified transformations on functional programs. In *Programming Languages and Systems - 25th European Symposium on Programming (ESOP 2016)* (2016), P. Thiemann, Ed.,

vol. 9632 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 752–779.

Full Language Library

This appendix gives the full specification for our example language library from Chapter 2.

A.1 Host Language H

$\mathbb{B}^H = \varnothing$

A.1.1 Syntax

$\mathscr{C}^H = \{s, e, ty, n, i, \Gamma, \gamma\}$

$\mathbb{C}^H$:

$$
\begin{array}{rcl}
s & ::= & \mathit{skip} \mid \mathit{decl}(n, ty, e) \mid \mathit{assign}(n, e) \mid \mathit{seq}(s, s) \mid \mathit{ifte}(e, s, s) \mid \mathit{while}(e, s) \\
e & ::= & \mathit{var}(n) \mid \mathit{intlit}(i) \mid \mathit{true} \mid \mathit{false} \mid \mathit{add}(e, e) \mid \mathit{eq}(e, e) \mid \mathit{gt}(e, e) \mid \mathit{not}(e) \\
ty & ::= & \mathit{int} \mid \mathit{bool} \\
\Gamma & ::= & \mathit{nilty} \mid \mathit{consty}(n, ty, \Gamma) \\
\gamma & ::= & \mathit{nilval} \mid \mathit{consval}(n, e, \gamma)
\end{array}
$$

A.1.2 Relations

$$
\begin{aligned}
\mathscr{R}^H = \{ & \mathit{lkpTy}(\Gamma^*, n, ty), \quad \mathit{notBoundTy}(\Gamma^*, n), \quad \mathit{lkpVal}(\gamma^*, n, e), \quad \mathit{value}(e^*), \quad \mathit{vars}(e^*, 2^n), \\
& \Gamma \vdash e^* : ty, \quad \Gamma \vdash s^*, \Gamma, \quad \gamma \vdash e^* \Downarrow e, \quad (\gamma,\ s^*) \Downarrow \gamma\}
\end{aligned}
$$

$\mathbb{R}^H$ contains the following rules:

$\boxed{lkpTy(\Gamma^*, n, ty)}$

$$\frac{}{lkpTy(consty(n, ty, \Gamma), n, ty)}\ \text{LT-Head} \qquad \frac{n \neq n' \qquad lkpTy(\Gamma, n, ty)}{lkpTy(consty(n', ty', \Gamma), n, ty)}\ \text{LT-Tail}$$

$\boxed{notBoundTy(\Gamma^*, n)}$

$$\frac{}{notBoundTy(nilty, n)}\ \text{NLT-Nil} \qquad \frac{n \neq n' \qquad notBoundTy(\Gamma, n)}{notBoundTy(consty(n', ty', \Gamma), n)}\ \text{NLT-Cons}$$

$\boxed{lkpVal(\gamma^*, n, e)}$

$$\frac{}{lkpVal((consval(n, v, \gamma)), n, v)}\ \text{LV-Head} \qquad \frac{n \neq n' \qquad lkpVal(\gamma, n, v)}{lkpVal((consval(n', v', \gamma)), n, v)}\ \text{LV-Tail}$$

$\boxed{value(e^*)}$

$$\frac{}{value(intlit(i))}\ \text{V-Int} \qquad \frac{}{value(true)}\ \text{V-True} \qquad \frac{}{value(false)}\ \text{V-False}$$

$\boxed{vars(e^*, 2^n)}$

$$\frac{}{vars(var(n), \{n\})}\ \text{VR-Var} \qquad \frac{}{vars(intlit(i), \varnothing)}\ \text{VR-Int}$$

$$\frac{}{vars(true, \varnothing)}\ \text{VR-True} \qquad \frac{}{vars(false, \varnothing)}\ \text{VR-False}$$

$$\frac{vars(e_1, vr_1) \qquad vars(e_2, vr_2)}{vars(add(e_1, e_2), (vr_1 \cup vr_2))}\ \text{VR-Add} \qquad \frac{vars(e_1, vr_1) \qquad vars(e_2, vr_2)}{vars(gt(e_1, e2), (vr_1 \cup vr_2))}\ \text{VR-Gt}$$

$$\frac{vars(e_1, vr_1) \qquad vars(e_2, vr_2)}{vars(eq(e_1, e_2), (vr_1 \cup vr_2))}\ \text{VR-Eq} \qquad \frac{vars(e, vr)}{vars(not(e), vr)}\ \text{VR-Not}$$

$\boxed{\Gamma \vdash e^* : ty}$

$$\frac{lkpTy(\Gamma, n, ty)}{\Gamma \vdash var(n) : ty}\ \text{T-Var} \qquad \frac{}{\Gamma \vdash true : bool}\ \text{T-True}$$

$$\frac{}{\Gamma \vdash intlit(i) : int}\ \text{T-Int} \qquad \frac{}{\Gamma \vdash false : bool}\ \text{T-False}$$

$$\frac{\Gamma \vdash e_1 : int \qquad \Gamma \vdash e_2 : int}{\Gamma \vdash add(e_1, e_2) : int}\ \text{T-Add} \qquad \frac{\Gamma \vdash e_1 : int \qquad \Gamma \vdash e_2 : int}{\Gamma \vdash gt(e_1, e_2) : bool}\ \text{T-Gt}$$

$$\frac{\Gamma \vdash e_1 : int \qquad \Gamma \vdash e_2 : int}{\Gamma \vdash eq(e_1, e_2) : bool}\ \text{T-Eq} \qquad \frac{\Gamma \vdash e : bool}{\Gamma \vdash not(e) : bool}\ \text{T-Not}$$

$\boxed{\Gamma \vdash s^*, \Gamma}$

$$\frac{}{\Gamma \vdash skip, \Gamma}\ \text{TS-Skip} \qquad \frac{\Gamma \vdash e : ty \qquad notBoundTy(\Gamma, n)}{\Gamma \vdash decl(n, ty, e), consty(n, ty, \Gamma)}\ \text{TS-Decl}$$

$$\frac{\Gamma \vdash s_1, \Gamma' \qquad \Gamma' \vdash s_2, \Gamma''}{\Gamma \vdash seq(s_1, s_2), \Gamma''}\ \text{TS-Seq} \qquad \frac{\Gamma \vdash e : ty \qquad lkpTy(\Gamma, n, ty)}{\Gamma \vdash assign(n, e), \Gamma}\ \text{TS-Assign}$$

$$\frac{\Gamma \vdash e : bool \qquad \Gamma \vdash s, \Gamma'}{\Gamma \vdash while(e, s), \Gamma}\ \text{TS-While} \qquad \frac{\Gamma \vdash e : bool \qquad \Gamma \vdash s_1, \Gamma' \qquad \Gamma \vdash s_1, \Gamma''}{\Gamma \vdash ifte(e, s_1, s_2), \Gamma}\ \text{TS-If}$$

$\boxed{\gamma \vdash e^* \Downarrow e}$

$$\frac{lkpVal(\gamma, n, v)}{\gamma \vdash var(n) \Downarrow v}\ \text{E-Var} \qquad \frac{}{\gamma \vdash true \Downarrow true}\ \text{E-True}$$

$$\frac{}{\gamma \vdash intlit(i) \Downarrow intlit(i)}\ \text{E-Int} \qquad \frac{}{\gamma \vdash false \Downarrow false}\ \text{E-False}$$

$$\frac{\gamma \vdash e \Downarrow false}{\gamma \vdash not(e) \Downarrow true}\ \text{E-Not-T} \qquad \frac{\gamma \vdash e \Downarrow true}{\gamma \vdash not(e) \Downarrow false}\ \text{E-Not-F}$$

$$\frac{\gamma \vdash e_1 \Downarrow \mathit{intlit}(i_1) \qquad \gamma \vdash e_2 \Downarrow \mathit{intlit}(i_2) \qquad \mathit{plus}(i_1, i_2, i)}{\gamma \vdash \mathit{add}(e_1, e_2) \Downarrow \mathit{intlit}(i)} \text{ E-Add}$$

$$\frac{\gamma \vdash e_1 \Downarrow v_1 \qquad \gamma \vdash e_2 \Downarrow v_2 \qquad v_1 = v_2}{\gamma \vdash \mathit{eq}(e_1, e_2) \Downarrow \mathit{true}} \text{ E-Eq-T}$$

$$\frac{\gamma \vdash e_1 \Downarrow v_1 \qquad \gamma \vdash e_2 \Downarrow v_2 \qquad v_1 \neq v_2}{\gamma \vdash \mathit{eq}(e_1, e_2) \Downarrow \mathit{false}} \text{ E-Eq-F}$$

$$\frac{\gamma \vdash e_1 \Downarrow \mathit{intlit}(i_1) \qquad \gamma \vdash e_2 \Downarrow \mathit{intlit}(i_2) \qquad i_1 > i_2}{\gamma \vdash \mathit{gt}(e_1, e_2) \Downarrow \mathit{true}} \text{ E-Gt-T}$$

$$\frac{\gamma \vdash e_1 \Downarrow \mathit{intlit}(i_1) \qquad \gamma \vdash e_2 \Downarrow \mathit{intlit}(i_2) \qquad i_1 \leq i_2}{\gamma \vdash \mathit{gt}(e_1, e_2) \Downarrow \mathit{false}} \text{ E-Gt-F}$$

$\boxed{(\gamma,\ s^*) \Downarrow \gamma}$

$$\frac{}{(\gamma,\ \mathit{skip}) \Downarrow \gamma} \text{ X-Skip} \qquad \frac{(\gamma,\ s_1) \Downarrow \gamma' \qquad (\gamma',\ s_2) \Downarrow \gamma''}{(\gamma,\ \mathit{seq}(s_1, s_2)) \Downarrow \gamma''} \text{ X-Seq}$$

$$\frac{\gamma \vdash e \Downarrow \mathit{true} \qquad (\gamma,\ s_1) \Downarrow \gamma'}{(\gamma,\ \mathit{ifte}(e, s_1, s_2)) \Downarrow \gamma'} \text{ X-If-T} \qquad \frac{\gamma \vdash e \Downarrow \mathit{false} \qquad (\gamma,\ s_2) \Downarrow \gamma'}{(\gamma,\ \mathit{ifte}(e, s_1, s_2)) \Downarrow \gamma'} \text{ X-If-F}$$

$$\frac{\gamma \vdash e \Downarrow v}{(\gamma,\ \mathit{decl}(n, \mathit{ty}, e)) \Downarrow \mathit{consval}(n, v, \gamma)} \text{ X-Decl} \qquad \frac{\gamma \vdash e \Downarrow v \qquad \mathit{update}(\gamma, n, v, \gamma')}{(\gamma,\ \mathit{assign}(n, e)) \Downarrow \gamma'} \text{ X-Assign}$$

$$\frac{\gamma \vdash e \Downarrow \mathit{false}}{(\gamma,\ \mathit{while}(e, s)) \Downarrow \gamma} \text{ X-While-F}$$

$$\frac{\gamma \vdash e \Downarrow \mathit{true} \qquad (\gamma,\ s) \Downarrow \gamma' \qquad (\gamma',\ \mathit{while}(e, s)) \Downarrow \gamma''}{(\gamma,\ \mathit{while}(e, s)) \Downarrow \gamma''} \text{ X-While-T}$$

A.1.3 Projections and Default Rules

$\mathscr{T}^H = \{e : proj_e(e, e), s : proj_s(s, s), ty : proj_{ty}(ty, ty)\}$

$\mathbb{T}^H = \varnothing$

$\mathbb{S}^H = \varnothing$

A.2 List Extension L

$\mathbb{B}^L = \{H\}$

A.2.1 Syntax

$\mathscr{C}^L = \varnothing$

$\mathbb{C}^L$:

$$
\begin{array}{lcl}
s & ::= & splitlist(n, n, e) \\
e & ::= & nil \mid cons(e, e) \mid null(e) \mid head(e) \mid tail(e) \\
ty & ::= & list(ty)
\end{array}
$$

A.2.2 Relations

$\mathscr{R}^L = \varnothing$

$\mathbb{R}^L$ contains the following rules:

$$\boxed{value(e^*)}$$

$$\frac{}{value(nil)}\ \text{V-Nil} \qquad \frac{value(e_1) \qquad value(e_2)}{value(cons(e_1, e_2))}\ \text{V-Cons}$$

$\boxed{vars(e^*, 2^n)}$

$$\frac{}{vars(nil, \varnothing)}\ \text{VR-Nil} \qquad \frac{vars(e, vr)}{vars(head(e), vr)}\ \text{VR-Head}$$

$$\frac{vars(e_1, vr_1) \qquad vars(e_2, vr_2)}{vars(cons(e_1, e_2), (vr_1 \cup vr_2))}\ \text{VR-Cons} \qquad \frac{vars(e, vr)}{vars(tail(e), vr)}\ \text{VR-Tail}$$

$$\frac{vars(e, vr)}{vars(null(e), vr)}\ \text{VR-Null}$$

$\boxed{\Gamma \vdash e^* : ty}$

$$\frac{}{\Gamma \vdash nil : list(ty)}\ \text{T-Nil} \qquad \frac{\Gamma \vdash e_1 : ty \qquad \Gamma \vdash e_2 : list(ty)}{\Gamma \vdash cons(e_1, e_2) : list(ty)}\ \text{T-Cons}$$

$$\frac{\Gamma \vdash e : list(ty)}{\Gamma \vdash head(e) : ty}\ \text{T-Head} \qquad \frac{\Gamma \vdash e : list(ty)}{\Gamma \vdash tail(e) : list(ty)}\ \text{T-Tail}$$

$$\frac{\Gamma \vdash e : list(ty)}{\Gamma \vdash null(e) : bool}\ \text{T-Null}$$

$\boxed{\Gamma \vdash s^*, \Gamma}$

$$\frac{\Gamma \vdash e : list(ty) \qquad lkpTy(\Gamma, n_{hd}, ty) \qquad lkpTy(\Gamma, n_{tl}, list(ty))}{\Gamma \vdash splitlist(n_{hd}, n_{tl}, e), \Gamma}\ \text{TS-Splitlist}$$

$\boxed{\gamma \vdash e^* \Downarrow e}$

$$\frac{}{\gamma \vdash \mathit{nil} \Downarrow \mathit{nil}}\ \text{E-Nil} \qquad \frac{\gamma \vdash e \Downarrow \mathit{cons}(v_1, v_2)}{\gamma \vdash \mathit{head}(e) \Downarrow v_1}\ \text{E-Head}$$

$$\frac{\gamma \vdash e_1 \Downarrow v_1 \qquad \gamma \vdash e_2 \Downarrow v_2}{\gamma \vdash \mathit{cons}(e_1, e_2) \Downarrow \mathit{cons}(v_1, v_2)}\ \text{E-Cons} \qquad \frac{\gamma \vdash e \Downarrow \mathit{cons}(v_1, v_2)}{\gamma \vdash \mathit{tail}(e) \Downarrow v_2}\ \text{E-Tail}$$

$$\frac{\gamma \vdash e \Downarrow \mathit{nil}}{\gamma \vdash \mathit{null}(e) \Downarrow \mathit{true}}\ \text{E-Null-T} \qquad \frac{\gamma \vdash e \Downarrow \mathit{cons}(v_1, v_2)}{\gamma \vdash \mathit{null}(e) \Downarrow \mathit{false}}\ \text{E-Null-F}$$

$\boxed{(\gamma,\ s^*) \Downarrow \gamma}$

$$\frac{\gamma \vdash e \Downarrow \mathit{cons}(v_1, v_2) \qquad n_{hd} \neq n_{tl} \qquad \mathit{update}(\gamma, n_{hd}, v_1, \gamma') \qquad \mathit{update}(\gamma', n_{tl}, v_2, \gamma'')}{(\gamma,\ \mathit{splitlist}(n_{hd}, n_{tl}, e)) \Downarrow \gamma''}\ \text{X-Splitlist}$$

A.2.3 Projections and Default Rules

$\mathscr{T}^L = \varnothing$

$\mathbb{S}^L = \varnothing$

$\mathbb{T}^L$ contains the following rules:

$$\frac{}{\mathit{proj}_e(\mathit{null}(e), e)}\ \text{P-Null} \qquad \frac{}{\mathit{proj}_e(\mathit{nil}, \mathit{true})}\ \text{P-Nil}$$

$$\frac{}{\mathit{proj}_e(\mathit{head}(e), e)}\ \text{P-Head} \qquad \frac{}{\mathit{proj}_e(\mathit{cons}(e_1, e_2), \mathit{eq}(e_1, e_2))}\ \text{P-Cons}$$

$$\frac{}{\mathit{proj}_e(\mathit{tail}(e), e)}\ \text{P-Tail} \qquad \frac{}{\mathit{proj}_{ty}(\mathit{list}(ty), ty)}\ \text{P-List}$$

$$\frac{n_{hd} \neq n_{tl}}{\begin{array}{l}\mathit{proj}_s(\mathit{splitlist}(n_{hd}, n_{tl}, e),\\ \quad \mathit{seq}(\mathit{seq}(\mathit{assign}(n_{hd}, e), \mathit{assign}(n_{tl}, \mathit{tail}(\mathit{var}(n_{hd})))),\\ \quad\quad \mathit{assign}(n_{hd}, \mathit{head}(\mathit{var}(n_{hd})))))\end{array}}\ \text{P-Splitlist}$$

A.3 Security Extension S

$\mathbb{B}^S = \{H\}$

A.3.1 Syntax

$\mathscr{C}^S = \{sl, \Sigma\}$

$\mathbb{C}^S$:

$$
\begin{array}{lcl}
s & ::= & secdecl(n, ty, sl, e) \\
sl & ::= & public \mid private \\
\Sigma & ::= & nilsec \mid conssec(n, sl, \Sigma)
\end{array}
$$

A.3.2 Relations

$\mathscr{R}^S = \{\ lkpSec(\Sigma^*, n, sl),\quad join(sl^*, sl, sl),\quad \Sigma \vdash level(e^*, sl),\quad \Sigma\ sl \vdash secure(s^*, \Sigma)\ \}$

$\mathbb{R}^S$ contains the following rules:

$\boxed{\Gamma \vdash s^*, \Gamma}$

$$\frac{\gamma \vdash e : ty \qquad notBoundTy(\Gamma, n)}{\Gamma \vdash secdecl(n, ty, sl, e), consty(n, ty, \Gamma)}\ \text{TS-SECDECL}$$

$\boxed{(\gamma,\ s^*) \Downarrow \gamma}$

$$\frac{\gamma \vdash e \Downarrow v}{(\gamma,\ secdecl(n, ty, sl, e)) \Downarrow consval(n, v, \gamma)}\ \text{X-SECDECL}$$

$\boxed{lkpSec(\Sigma^*, n, sl)}$

$$\frac{}{lkpSec(conssec(n, sl, \Sigma), n, sl)}\ \text{LS-Head} \qquad \frac{n \neq n' \qquad lkpSec(\Sigma, n, sl)}{lkpSec(conssec(n', sl', \Sigma), n, sl)}\ \text{LS-Tail}$$

$\boxed{join(sl^*, sl, sl)}$

$$\frac{}{join(public, public, public)}\ \text{J-Public}$$

$$\frac{}{join(private, \ell, private)}\ \text{J-Private-L} \qquad \frac{}{join(\ell, private, private)}\ \text{J-Private-R}$$

$\boxed{\Sigma \vdash level(e^*, sl)}$

$$\frac{lkpSec(\Sigma, n, \ell)}{\Sigma \vdash level(var(n), \ell)}\ \text{L-Var} \qquad \frac{}{\Sigma \vdash level(intlit(i), public)}\ \text{L-Int}$$

$$\frac{}{\Sigma \vdash level(true, public)}\ \text{L-True} \qquad \frac{}{\Sigma \vdash level(false, public)}\ \text{L-False}$$

$$\frac{\Sigma \vdash level(e, \ell)}{\Sigma \vdash level(not(e), \ell)}\ \text{L-Not}$$

$$\frac{\Sigma \vdash level(e_1, \ell_1) \qquad \Sigma \vdash level(e_2, \ell_2) \qquad join(\ell_1, \ell_2, \ell)}{\Sigma \vdash level(add(e_1, e_2), \ell)}\ \text{L-Add}$$

$$\frac{\Sigma \vdash level(e_1, \ell_1) \qquad \Sigma \vdash level(e_2, \ell_2) \qquad join(\ell_1, \ell_2, \ell)}{\Sigma \vdash level(gt(e_1, e_2), \ell)}\ \text{L-Gt}$$

$$\frac{\Sigma \vdash level(e_1, \ell_1) \qquad \Sigma \vdash level(e_2, \ell_2) \qquad join(\ell_1, \ell_2, \ell)}{\Sigma \vdash level(eq(e_1, e_2), \ell)}\ \text{L-Eq}$$

$\boxed{\Sigma\ sl \vdash secure(s^*, \Sigma)}$

$$\frac{}{\Sigma\ \ell \vdash secure(skip, \Sigma)}\ \text{S-Skip}$$

$$\frac{\Sigma\ \ell \vdash secure(s_1, \Sigma') \qquad \Sigma'\ \ell \vdash secure(s_2, \Sigma'')}{\Sigma\ \ell \vdash secure(seq(s_1, s_2), \Sigma'')}\ \text{S-Seq}$$

$$\frac{\Sigma \vdash level(e, public)}{\Sigma\ public \vdash secure(decl(n, ty, e), conssec(n, public, \Sigma))}\ \text{S-Decl}$$

$$\frac{\Sigma \vdash level(e, \ell) \qquad lkpSec(\Sigma, n, private)}{\Sigma\ \ell' \vdash secure(assign(n, e), \Sigma)}\ \text{S-Assign-Private}$$

$$\frac{\Sigma \vdash level(e, public) \qquad lkpSec(\Sigma, n, public)}{\Sigma\ public \vdash secure(assign(n, e), \Sigma)}\ \text{S-Assign-Public}$$

$$\frac{\Sigma \vdash level(e, \ell) \qquad join(\ell', \ell, \ell'') \qquad \Sigma\ \ell'' \vdash secure(s_1, \Sigma_1) \qquad \Sigma\ \ell'' \vdash secure(s_2, \Sigma_2)}{\Sigma\ \ell' \vdash secure(ifte(e, s_1, s_2), \Sigma)}\ \text{S-If}$$

$$\frac{\Sigma \vdash level(e, \ell) \qquad join(\ell', \ell, \ell'') \qquad \Sigma\ \ell'' \vdash secure(s, \Sigma')}{\Sigma\ \ell' \vdash secure(while(e, s), \Sigma)}\ \text{S-While}$$

$$\frac{\Sigma \vdash level(e, \ell)}{\Sigma\ \ell' \vdash secure(secdecl(n, ty, private, e), conssec(n, private, \Sigma))}\ \text{S-Secdecl-Private}$$

$$\frac{\Sigma \vdash level(e, public)}{\Sigma\ public \vdash secure(secdecl(n, ty, public, e), conssec(n, public, \Sigma))}\ \text{S-Secdecl-Public}$$

A.3.3 Projections and Default Rules

$\mathscr{T}^S = \{sl : proj_{sl}(sl, sl)\}$

$\mathbb{T}^S$ contains the following rule:

$$\frac{}{proj_s(ns, secdecl(n, ty, sl, e), decl(n, ty, e))}\ \text{P-Secdecl}$$

$\mathbb{S}^S$ contains the following rules:

$$\frac{proj_e(e, e') \qquad \Sigma \vdash level(e', \ell)}{\Sigma \vdash level(e, \ell)}\ \text{L-Default}$$

$$\frac{proj_s(s, s') \qquad \Sigma\ \ell \vdash secure(s', \Sigma')}{\Sigma\ \ell \vdash secure(s, \Sigma')}\ \text{S-Default}$$

A.4 Optimization Extension O

$\mathbb{B}^O = \{H\}$

A.4.1 Syntax

$\mathscr{C}^O = \varnothing$

$\mathbb{C}^O = \varnothing$

A.4.2 Relations

$\mathscr{R}^O = \{\ opt_e(e^*, e), \quad opt_s(s^*, s), \quad notInt(e^*), \quad notBool(e^*)\ \}$

$\mathbb{R}^O$ contains the following rules:

$\boxed{opt_e(e^*, e)}$

$$\frac{}{opt_e(var(n), var(n))}\ \text{OE-Var} \qquad \frac{}{opt_e(intlit(i), intlit(i))}\ \text{OE-Int}$$

$$\frac{}{opt_e(true, true)}\ \text{OE-True} \qquad \frac{}{opt_e(false, false)}\ \text{OE-False}$$

$$\frac{opt_e(e_1, intlit(i_1)) \qquad opt_e(e_2, intlit(i_2)) \qquad plus(i_1, i_2, i)}{opt_e(add(e_1, e_2), intlit(i))}\ \text{OE-Add-I}$$

$$\frac{opt_e(e_1, e_1') \qquad opt_e(e_2, e_2') \qquad notInt(e_1')}{opt_e(add(e_1, e_2), add(e_1', e_2'))}\ \text{OE-Add-O-1}$$

$$\frac{opt_e(e_1, e_1') \qquad opt_e(e_2, e_2') \qquad notInt(e_2')}{opt_e(add(e_1, e_2), add(e_1', e_2'))}\ \text{OE-Add-O-2}$$

$$\frac{opt_e(e_1, e) \qquad opt_e(e_2, e)}{opt_e(eq(e_1, e_2), true)}\ \text{OE-Eq-T}$$

$$\frac{opt_e(e_1, e_1') \qquad opt_e(e_2, e_2') \qquad value(e_1') \qquad value(e_2') \qquad e_1' \neq e_2'}{opt_e(eq(e_1, e_2), false)}\ \text{OE-Eq-F}$$

$$\frac{opt_e(e_1, e_1') \qquad opt_e(e_2, e_2') \qquad \neg value(e_1') \qquad e_1' \neq e_2'}{opt_e(eq(e_1, e_2), eq(e_1', e_2'))}\ \text{OE-Eq-O-1}$$

$$\frac{opt_e(e_1, e_1') \qquad opt_e(e_2, e_2') \qquad \neg value(e_2') \qquad e_1' \neq e_2'}{opt_e(eq(e_1, e_2), eq(e_1', e_2'))}\ \text{OE-Eq-O-2}$$

$$\frac{opt_e(e_1, intlit(i_1)) \qquad opt_e(e_2, intlit(i_2)) \qquad greaterThan(i_1, i_2)}{opt_e(gt(e_1, e_2), true)}\ \text{OE-Gt-T}$$

$$\frac{opt_e(e_1, intlit(i_1)) \qquad opt_e(e_2, intlit(i_2)) \qquad lessEq(i_1, i_2)}{opt_e(gt(e_1, e_2), false)}\ \text{OE-Gt-F}$$

$$\frac{opt_e(e_1, e_1') \qquad opt_e(e_2, e_2') \qquad notInt(e_1')}{opt_e(gt(e_1, e_2), gt(e_1', e_2'))}\ \text{OE-Gt-O-1}$$

$$\frac{opt_e(e_1, e_1') \qquad opt_e(e_2, e_2') \qquad notInt(e_2')}{opt_e(gt(e_1, e_2), gt(e_1', e_2'))}\ \text{OE-Gt-O-2}$$

$$\frac{opt_e(e, false)}{opt_e(not(e), true)}\ \text{OE-Not-T} \qquad \frac{opt_e(e, true)}{opt_e(not(e), false)}\ \text{OE-Not-F}$$

$$\frac{opt_e(e, e') \qquad notBool(e')}{opt_e(not(e), not(e'))}\ \text{OE-Not-O}$$

$\boxed{opt_s(2^n, s^*)s}$

$$\frac{}{opt_s(skip, skip)}\ \text{OS-Skip} \qquad \frac{opt_e(e, e')}{opt_s(decl(n, ty, e), decl(n, ty, e'))}\ \text{OS-Decl}$$

$$\frac{opt_s(s_1, s_1') \qquad opt_s(s_2, s_2')}{opt_s(seq(s_1, s_2), seq(s_1', s_2'))}\ \text{OS-Seq} \qquad \frac{opt_e(e, e')}{opt_s(assign(n, e), assign(n, e'))}\ \text{OS-Assign}$$

$$\frac{opt_e(c, false)}{opt_s(while(c, b), skip)}\ \text{OS-While-F} \qquad \frac{opt_e(c, c') \qquad opt_s(b, b') \qquad c' \neq false}{opt_s(while(c, b), while(c', b'))}\ \text{OS-While-O}$$

$$\frac{opt_e(c, true) \qquad opt_s(t, t')}{opt_s(ifte(c, t, f), t')}\ \text{OS-If-T} \qquad \frac{opt_e(c, false) \qquad opt_s(f, f')}{opt_s(ifte(c, t, f), f')}\ \text{OS-If-F}$$

$$\frac{opt_e(c, c') \qquad notBool(c') \qquad opt_s(t, t') \qquad opt_s(f, f')}{opt_s(ifte(c, t, f), ifte(c', t', f'))}\ \text{OS-If-O}$$

$\boxed{notInt(e^*)}$

$$\frac{}{notInt(var(n))}\ \text{NI-Var} \qquad \frac{}{notInt(true)}\ \text{NI-True} \qquad \frac{}{notInt(false)}\ \text{NI-False}$$

$$\frac{}{notInt(add(e_1, e_2))}\ \text{NI-Add} \qquad \frac{}{notInt(eq(e_1, e_2))}\ \text{NI-Eq} \qquad \frac{}{notInt(gt(e_1, e_2))}\ \text{NI-Gt}$$

$$\frac{}{notInt(not(e))}\ \text{NI-Not}$$

$\boxed{notBool(e^*)}$

$$\frac{}{notBool(var(n))}\ \text{NB-Var} \qquad \frac{}{notBool(intlit(i))}\ \text{NB-Int} \qquad \frac{}{notBool(add(e_1, e_2))}\ \text{NB-Add}$$

$$\frac{}{notBool(eq(e_1, e_2))}\ \text{NB-Eq} \qquad \frac{}{notBool(gt(e_1, e_2))}\ \text{NB-Gt} \qquad \frac{}{notBool(not(e))}\ \text{NB-Not}$$

A.4.3 Projections and Default Rules

$\mathscr{T}^O = \varnothing$

$\mathbb{T}^O = \varnothing$

$\mathbb{S}^O$ contains the following rules:

$$\frac{}{opt_e(e,e)}\ \text{OE-Default} \qquad \frac{proj_s(ns,s,s') \qquad opt_s(ns,s')s''}{opt_s(ns,s)s''}\ \text{OS-Default}$$

$$\frac{}{notInt(e)}\ \text{NI-Default} \qquad \frac{}{notBool(e)}\ \text{NB-Default}$$

A.5 Dummy Composition Module D

$\mathbb{B}^D = \{H, L, S, O\}$

$\mathscr{C}^D = \varnothing$

$\mathbb{C}^D = \varnothing$

$\mathscr{R}^D = \varnothing$

$\mathbb{R}^D = \varnothing$

$\mathscr{T}^D = \varnothing$

$\mathbb{T}^D = \varnothing$

$\mathbb{S}^D = \varnothing$

Appendix B

Term Replacement, Substitution, and Unification

In our proofs in Chapter 4, we need some facts about the relationship between unification and term replacement (Definition 4.11) to show the proofs written as part of modular proofs can be used to prove the same properties in a larger composed language. We prove such facts here.

First, we show term replacement distributes over substitution.

Theorem B.1 (Term replacement distributes over substitution)**.** *Let c and d be constructors and $c(\overline{t})$ be a term. Let θ be a substitution whose domain does not contain any of the variables that appear in $\overline{t}$. Then*

1. *for any term s, $(s[\![c(\overline{t})/d]\!])[\theta[\![c(\overline{t})/d]\!]] = s[\theta][\![c(\overline{t})/d]\!]$.*
2. *for any formula F, $(F[\![c(\overline{t})/d]\!])[\theta[\![c(\overline{t})/d]\!]] = F[\theta][\![c(\overline{t})/d]\!]$.*
3. *for any unification problem $\mathcal{U}$, $(\mathcal{U}[\![c(\overline{t})/d]\!])[\theta[\![c(\overline{t})/d]\!]] = \mathcal{U}[\theta][\![c(\overline{t})/d]\!]$.*

Proof. To prove the first clause, we proceed by induction on the structure of s.

If s is a variable, it may or may not be in the domain of θ. If it is not, it remains the same variable on either side of the equation. If it is in the domain, it is replaced by $r[\![c(\overline{t})/d]\!]$ on the left and r on the right, which is then term replaced so both sides are equal.

If s is a term built by d, it is replaced by $c(\overline{t})$ on the left, which is then unchanged by the substitution because the variables in $\overline{t}$ are not in the domain of θ. On the right, it is still built by d after substitution, and thus is replaced with $c(\overline{t})$, making both sides equal.

If s is a term built by some other constructor, it has the form $c'(\overline{r})$ where $\overline{r}$ might be empty. For each r_i in $\overline{r}$, we have $(r_i[\![c(\overline{t})/d]\!])[\theta[\![c(\overline{t})/d]\!]] = r_i[\theta][\![c(\overline{t})/d]\!]$. Because substitution

and term replacement on a compound term are defined by applying the same operation to each sub-term, this then means we have the same results for the overall substitutions and term replacements, completing the proof for the first clause.

The clauses for formulas and unification problems clearly follow from the first. ■

Next, we prove term replacement is its own inverse if the term being substituted for a constructor is built by a constructor not occurring in the original and the constructor for which we are substituting does not take any arguments.

Lemma B.2 (Term replacement inversion)**.** *Let c and d be constructors where d does not take any arguments (a 0-ary constructor) and let $c(\bar{t})$ be a term.*

1. *Let s be a term in which c does not appear. Then $s[\![c(\bar{t})/d]\!][\![d/c]\!] = s$.*
2. *Let F be a formula in which c does not appear. Then $F[\![c(\bar{t})/d]\!][\![d/c]\!] = F$.*
3. *Let $\mathcal{U}$ be a unification problem in which c does not appear. Then $\mathcal{U}[\![c(\bar{t})/d]\!][\![d/c]\!] = \mathcal{U}$.*
4. *Let θ be a substitution in which c does not appear. Then $\theta[\![c(\bar{t})/d]\!][\![d/c]\!] = \theta$.*

Proof. To prove the first clause, we proceed by induction on the structure of s. If s is a variable, it is not changed by the term replacement. If s is d, $d[\![c(\bar{t})/d]\!] = c(\bar{t})$. Then replacing terms built by c with d ($c(\bar{t})[\![d/c]\!]$) gives us back d. If s is built by some other constructor, we carry out the term replacement on its arguments. This constructor cannot be c by assumption, so the second replacement of d for terms built by c also leaves the root constructor, replacing only in the sub-terms. Then, by the induction hypothesis, the sub-terms are the same after the second replacement as originally, and then the whole term is the same.

The clauses for formulas, unification problems, and substitutions clearly follow from the clause for terms. ■

We can also prove term replacement removes a constructor completely.

Lemma B.3 (Term replacement removes constructor)**.** *Let c and d be constructors and $c(\bar{t})$ be a term where $c(\bar{t})$ does not contain d.*

1. *Let s be a term. Then $s[\![c(\bar{t})/d]\!]$ does not contain d.*

2. *Let F be a formula. Then $F[\![c(\bar{t})/d]\!]$ does not contain d.*

3. *Let $\mathcal{U}$ be a unification problem. Then $\mathcal{U}[\![c(\bar{t})/d]\!]$ does not contain d.*

4. *Let θ be a substitution. Then $\theta[\![c(\bar{t})/d]\!]$ does not contain d.*

Proof. We can prove the first clause by induction on the structure of s. Whenever we come to a term built by d, we replace it with $c(\bar{t})$. Since $c(\bar{t})$ does not contain d, we eliminate each occurrence.

The clauses for formulas, unification problems, and substitutions clearly follow from the clause for terms. ■

The following algorithm, due to Martelli and Montanari [24], is guaranteed to terminate, either finding an mgu if the original problem is solvable, or failing if there is no unifier.

Definition B.4 (Unification algorithm)**.** *Repeatedly perform the actions of one of the rules in this list to find a unifier for a unification problem:*

- ***Reorder:*** *Select a pair $\langle t, x\rangle$ where t is not a variable and x is and replace it with $\langle x, t\rangle$.*
- ***Drop:*** *Select a pair $\langle x, x\rangle$ where x is a variable and drop it.*
- ***Variable elimination:*** *Select a pair $\langle x, t\rangle$ where x is a variable occurring elsewhere in the unification problem and where $t \neq x$ and x is not in t. Apply the substitution $\{\langle x, t\rangle\}$ to the rest of the problem.*
- ***Term reduction:*** *Select a pair $\langle c'(t_1, \ldots, t_n), c'(s_1, \ldots, s_n)\rangle$. Replace this with the pairs $\{\langle t_1, s_1\rangle, \ldots, \langle t_n, s_n\rangle\}$.*

If none of the rules apply and there is a pair not of the form $\langle x, t\rangle$ where t does not contain x, the unification has failed. If it succeeds, we say it is in solved form*, giving us a substitution.*

We will need a lemma about this algorithm, that pairs of the same term do not affect its result.

Lemma B.5 (Unification of identical pairs)**.** *Let $\mathcal{U}$ be a unification problem and t be a term. Let θ be an mgu for $\mathcal{U} \cup \{\langle t, t\rangle\}$ by the algorithm in Definition B.4. Then θ is also an mgu for $\mathcal{U}$ found by the algorithm.*

Proof. We proceed by induction on the steps taken in the algorithm to find the mgu for $\mathcal{U} \cup \{\langle t, t\rangle\}$.

If a step operates on a pair other than $\langle t, t\rangle$, the same step applies to the same pair in $\mathcal{U}$.

If a step operates on $\langle t, t\rangle$, it must either use the **drop** rule or the **term reduction** rule. If it uses the **drop** rule, the remainder of the two problems are the same and have the same unifier. If it uses the **term reduction** rule, t has the form $c(s_1, \ldots, s_n)$, and we get new pairs $\{\langle s_1, s_1\rangle, \ldots, \langle s_n, s_n\rangle\}$. We can then apply the induction hypothesis to eliminate each of these pairs, finishing the proof. ■

We can prove an mgu for a unification problem can be turned into one for the unification problem after term replacement by carrying out term replacement on it as well.

Theorem B.6 (Most general unifier after term replacement)**.** *Let c and d be constructors and $c(\bar{t})$ be a term. Let $\mathcal{U}$ be a unifiable unification problem where c and the variables in $\bar{t}$ do not occur. Further assume every occurrence of d in $\mathcal{U}$ has the same arguments, that is, every term in $\mathcal{U}$ with d as its top-level symbol is equal to every other term with d as its top-level symbol. Then there is an mgu θ for $\mathcal{U}$ such that $\theta[\![c(\bar{t})/d]\!]$ is an mgu for $\mathcal{U}[\![c(\bar{t})/d]\!]$.*

Proof. We will prove that if a unification problem $\mathcal{U}'$, in which c and the variables in $\bar{t}$ do not occur and all terms built by d are equal, steps to $\mathcal{U}''$ using the rules of the algorithm in Definition B.4, then $\mathcal{U}'[\![c(\bar{t})/d]\!]$ can step to $\mathcal{U}''[\![c(\bar{t})/d]\!]$ by a finite number of steps.

If $\mathcal{U}'$ steps by the **reorder** or **drop** rules, it is clear $\mathcal{U}'[\![c(\bar{t})/d]\!]$ also steps by the same rule because term replacement leaves variables unchanged.

If $\mathcal{U}'$ steps by the **variable elimination** rule with a pair $\langle x, s\rangle$, the pair $\langle x, s[\![c(\bar{t})/d]\!]\rangle$ is in $\mathcal{U}'[\![c(\bar{t})/d]\!]$. Because x does not occur in s and the variables in $\bar{t}$ do not occur in $\mathcal{U}'$, it is clear x does not occur $s[\![c(\bar{t})/d]\!]$ either. If x occurs elsewhere in $\mathcal{U}'[\![c(\bar{t})/d]\!]$, we can use

the same rule to step to $(\mathcal{U}'[\![c(\bar{t})/d]\!])[\{\langle x, s[\![c(\bar{t})/d]\!]\rangle\}]$. By Theorem B.1, this is the same as $(\mathcal{U}'[\{\langle x, s\rangle\}])[\![c(\bar{t})/d]\!]$, or $\mathcal{U}''[\![c(\bar{t})/d]\!]$. If x does not occur elsewhere, as it may only have occurred in $\mathcal{U}'$ in arguments to terms built by d, $\mathcal{U}'[\![c(\bar{t})/d]\!]$ and $\mathcal{U}''[\![c(\bar{t})/d]\!]$ are the same, so we take zero steps.

If $\mathcal{U}'$ steps by the **term reduction** rule with a pair $\langle c'(r_1, \ldots, r_n), c'(s_1, \ldots, s_n)\rangle$, consider whether c' is actually d. If it is, then the pair is actually $\langle d(r_1, \ldots, r_n), d(r_1, \ldots, r_n)\rangle$ because all terms with d as their top-level symbol are equal. The corresponding pair in $\mathcal{U}'[\![c(\bar{t})/d]\!]$ is $\langle c(\bar{t}), c(\bar{t})\rangle$. By Lemma B.5, we get the same eventual solved form with or without the pairs $\{\langle r_1, r_1\rangle, \ldots, \langle r_n, r_n\rangle\}$ created by applying the rule in solving $\mathcal{U}'$. We can similarly apply the term reduction rule in $\mathcal{U}'[\![c(\bar{t})/d]\!]$ and Lemma B.5 to eliminate all the pairs created from it. In both cases, this leaves us with the original problems without this pair, so the relation still holds. If c' is distinct from d, $\mathcal{U}''$ contains the pairs $\{\langle r_1, s_1\rangle, \ldots, \langle r_n, s_n\rangle\}$. We can also apply the term reduction rule to the corresponding pair in $\mathcal{U}'[\![c(\bar{t})/d]\!]$, giving us new pairs $\{\langle r_1[\![c(\bar{t})/d]\!], s_1[\![c(\bar{t})/d]\!]\rangle, \ldots, \langle r_n[\![c(\bar{t})/d]\!], s_n[\![c(\bar{t})/d]\!]\rangle\}$, so we have the new steps are $\mathcal{U}''$ and $\mathcal{U}''[\![c(\bar{t})/d]\!]$.

If $\mathcal{U}'$ is in solved form, we have $\theta = \mathcal{U}'$, and $\mathcal{U}'$ only contains pairs of the form $\langle x, s\rangle$ where x does not occur in s. Because term substitution does not affect variables, $\mathcal{U}'[\![c(\bar{t})/d]\!]$ is also only pairs $\langle x, s[\![c(\bar{t})/d]\!]\rangle$. Furthermore, because x does not occur in s and the variables in $\bar{t}$ do not occur in $\mathcal{U}'$, x also does not occur in any $s[\![c(\bar{t})/d]\!]$. Then $\mathcal{U}'[\![c(\bar{t})/d]\!]$ is also in solved form, giving us an mgu $\theta[\![c(\bar{t})/d]\!]$. ■

We can now show a unification problem having an mgu after term replacement also implies it had one before term replacement.

Theorem B.7 (Most general unifier before term replacement)**.** *Let c and d be constructors where d does not take any arguments (a 0-ary constructor) and let $c(\bar{t})$ be a term where d does not occur in $c(\bar{t})$ Let $\mathcal{U}$ be a unification problem where c and the variables in $\bar{t}$ do not occur. If there is an mgu for $\mathcal{U}[\![c(\bar{t})/d]\!]$, then there is an mgu θ for $\mathcal{U}$ such that $\theta[\![c(\bar{t})/d]\!]$ is an mgu for $\mathcal{U}[\![c(\bar{t})/d]\!]$.*

Proof. For simplicity's sake, we will refer to $\mathcal{U}[\![c(\bar{t})/d]\!]$ as $\mathcal{U}'$. By Lemma B.2, we have that

$\mathcal{U}[\![c(\bar{t})/d]\!][\![d/c]\!] = \mathcal{U}$, that is, $\mathcal{U}'[\![d/c]\!] = \mathcal{U}$. Because the only occurrences of c in $\mathcal{U}'$ are the ones coming from replacing d with $c(\bar{t})$, as we assumed $\mathcal{U}$ did not have c in it, all occurrences of c have the same arguments. We also know d does not appear in $\mathcal{U}'$ by Lemma B.3. Then, assuming there is an mgu for $\mathcal{U}'$, we can apply Theorem B.6 to show there is an mgu θ for $\mathcal{U}'$ and an mgu $\theta[\![d/c]\!]$ for $\mathcal{U}'[\![d/c]\!]$. But this unification problem is simply $\mathcal{U}$, so we have an mgu $\theta[\![d/c]\!]$ for $\mathcal{U}$. By using Theorem B.6 again, this time with $\mathcal{U}$ as the unification problem, we get an mgu θ' for $\mathcal{U}$ such that $\theta'[\![c(\bar{t})/d]\!]$ is an mgu for $\mathcal{U}[\![c(\bar{t})/d]\!]$, completing the proof. ■