The Dissertation Committee for Benjamin James Delaware
certifies that this is the approved version of the following dissertation:

Feature modularity in mechanized reasoning

Committee:

William R. Cook, Supervisor

Don Batory

Adam Chlipala

Warren A. Hunt

Keshav Pingali

Feature modularity in mechanized reasoning

by

Benjamin James Delaware, B.S., B.A., M.Sc.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2013

To Shannon and Sydney

Acknowledgments

Without the support of numerous people over the past seven years, I never would have completed this dissertation. My advisor William Cook afforded me the freedom to explore and then provided clarity to my discoveries. My shadow advisor Don Batory has been a tireless source of encouragement and insightful questions. The comments and suggestions of my committee members, Adam Chlipala, Warren Hunt, and Keshav Pingali were invaluable in crafting this dissertation. The conversations with my collaborators Bruno C. d. S. Oliveira and Tom Schrijvers have always been illuminating, even though we were rarely on the same continent. I am grateful for the many ways in which my friends in Austin have contributed to this dissertation. We never read a paper, but the discussions with my fellow Formal Methods Hoedown members Donald Nguyen, Ian Wehrman, and Ricky Chang were always a delight. I swam countless laps with the UTCS Swim Team, Yinon Bentor and Ann Kilzer— someday one of the brilliant ideas I have while swimming will hold up on dry land. Saturdays frolfing with Thomas Finsterbusch, Joel Hestness, and Andrew Matsuoka taught me the value of both patience and breakfast tacos. Finally, I am indebted to everyone else who I have been lucky enough to call friend during my time in Austin.

Benjamin James Delaware

The University of Texas at Austin

December 2013

Feature modularity in mechanized reasoning

Publication No. ______________________

Benjamin James Delaware, Ph.D.
The University of Texas at Austin, 2013

Supervisor: William R. Cook

Complex systems are naturally understood as combinations of their distinguishing characteristics or *features*. Distinct features differentiate between variations of configurable systems and also identify the novelties of extensions. The implementation of a conceptual feature is often scattered throughout an artifact, forcing designers to understand the entire artifact in order to reason about the behavior of a single feature. It is particularly challenging to independently develop novel extensions to complex systems as a result.

This dissertation shows how to modularly reason about the implementation of conceptual features in both the formalizations of programming languages and object-oriented software product lines. In both domains, modular verification of features can be leveraged to reason about the behavior of artifacts in which they are included: fully mechanized metatheory proofs for programming languages can be synthesized from independently developed proofs, and programs built from well-formed feature modules are guaranteed to be well-formed without needing to be typechecked. Modular reasoning about individual features can furthermore be used to efficiently reason about families of languages and programs which share a common set of features.

Contents

Chapter 1

Introduction

Every program can be described by the functionality or features it provides. The features offered by the Linux kernel, for example, include support for several different file systems and processor types, optional networking support and any number of device drivers. Users can select desired features before compilation in order to tailor the installation to their specific needs. In this regard, there is no single Linux kernel — it is a family of related programs whose individual members are deployed on machines worldwide.

The design of any complex system can be similarly identified by its distinguishing characteristics or *features*. Features provide a natural means to understand the variations of configurable systems, as with the Linux kernel. The evolution of an existing system is often described in terms of the new features it offers: the iPhone 5s, for example, is an iPhone 5 plus a fingerprint sensor and a coat of gold paint, while Java 5.0 is Java 1.4[1] plus generics, annotations, and enumerations.

A feature's implementation is typically scattered throughout a system, making it difficult to generate variations which remove or modify features and to independently develop and combine extensions. A number of tools have been developed to help modularize a feature's implementation into distinct components which can be composed to synthesize

[1]The release numbering was changed with this version of Java- its internal number was actually 1.5.

a product with a specified combination of features [Bat04, AKL09]. A fundamental and open problem is how to modularly reason about semantic properties of these components. The cross-cutting nature of features complicates reasoning, making it particularly challenging to independently develop novel extensions, as designers must understand the (possibly quite complex) existing system in order to reason about the impact of a new feature. This dissertation addresses the problem of semantic feature modularity in both the formalization of programming language metatheory and software development. It furthermore demonstrates how modular reasoning enables efficient reasoning about a family of products which share a common set of features.

1.1 Challenges and Contributions

To understand the challenges and benefits of modularly reasoning about features, consider the design and implementation of a simple programming language with arithmetic and boolean expressions. A similar language might include arithmetic expressions and anonymous functions built from lambda abstractions. The high-level similarities between the two languages are captured by their common feature, arithmetic expressions, while their differences are expressed by the features they do not share, boolean and lambda expressions.

1.1.1 Feature Modularity in Programming Language Metatheory

Tasked with creating a language with arithmetic and boolean expressions, a programming language researcher will probably provide a formal description of the syntax, type system, and semantics of the language, as shown in Figure 1.1. The *syntax* describes the set of valid expressions using Backus Normal Form [Con79]. The type system assigns a type T to an expression e, $\vdash \mathsf{e} : \mathsf{T}$, while the dynamic semantics show how an expression e evaluates to a value v, $\mathsf{e} \Downarrow \mathsf{v}$. The latter two facets of a language are typically specified as a set of declarative rules [Pie02] meant to guide the implementation of the language.

To complete the formalization, the programming language researcher will typically write some proofs about its *metatheory* or the behavior of its type system and semantics. Most often, this is a proof that the type system is *sound*— that "well-typed programs won't 'go wrong"' [Mil78]. Type soundness for the language of Figure 1.1, for example, is defined as:

Theorem 1.1.1. *Any expression* e *which has type* T, $\vdash \mathsf{e} : \mathsf{T}$, *will reduce to a value* v, $\mathsf{e} \Downarrow \mathsf{v}$, *which also has type* T, $\vdash \mathsf{v} : \mathsf{T}$.

Traditionally, metatheory proofs of soundness properties like Theorem 1.1.1 are *informal*— convincing arguments written in English.

The direct connection between the expression language features and their formal definitions makes it easy to synthesize a formal description from a specification. The language with arithmetic and boolean features, for example, is the combination of the definitions in the

Arith

$$\mathsf{e} \; ::= \; \mathbb{N} \; \mid \; \mathsf{e} + \mathsf{e}$$

$$\frac{\mathsf{n} \in \mathbb{N}}{\vdash \mathsf{n} : \mathsf{natT}} \; (\text{T-NLit})$$

$$\frac{\vdash \mathsf{e_m} : \mathsf{natT} \qquad \vdash \mathsf{e_n} : \mathsf{natT}}{\vdash \mathsf{e_m} + \mathsf{e_n} : \mathsf{natT}} \; (\text{T-Plus})$$

$$\frac{}{\mathsf{n} \Downarrow \mathsf{n}} \; (\text{R-NLit})$$

$$\frac{\vdash \mathsf{e_m} \Downarrow \mathsf{m} \qquad \vdash \mathsf{e_n} \Downarrow \mathsf{n}}{\vdash \mathsf{e_m} + \mathsf{e_n} \Downarrow \mathsf{m} + \mathsf{n}} \; (\text{R-Plus})$$

Bool

$$\mathsf{e} \; ::= \; \mathbb{B} \; \mid \; \textbf{if } \mathsf{e} \textbf{ then } \mathsf{e} \textbf{ else } \mathsf{e}$$

$$\frac{\mathsf{b} \in \mathbb{B}}{\vdash \mathsf{b} : \mathsf{boolT}} \; (\text{R-BLit})$$

$$\frac{\vdash \mathsf{i} : \mathsf{boolT} \qquad \vdash \mathsf{t} : \mathsf{T} \qquad \vdash \mathsf{e} : \mathsf{T}}{\vdash \textbf{if } \mathsf{i} \textbf{ then } \mathsf{t} \textbf{ else } \mathsf{e} : \mathsf{T}} \; (\text{T-Cond})$$

$$\frac{}{\vdash \mathsf{b} \Downarrow \mathsf{b}} \; (\text{T-BLit})$$

$$\frac{\vdash \mathsf{e_i} \Downarrow \mathsf{true} \qquad \vdash \mathsf{e_t} \Downarrow \mathsf{v_t}}{\vdash \textbf{if } \mathsf{e_i} \textbf{ then } \mathsf{e_t} \textbf{ else } \mathsf{e_e} \Downarrow \mathsf{v_t}} \; (\text{R-Cond-T})$$

$$\frac{\vdash \mathsf{e_i} \Downarrow \mathsf{false} \qquad \vdash \mathsf{e_e} \Downarrow \mathsf{v_e}}{\vdash \textbf{if } \mathsf{e_i} \textbf{ then } \mathsf{e_t} \textbf{ else } \mathsf{e_e} \Downarrow \mathsf{v_t}} \; (\text{R-Cond-E})$$

Figure 1.1: Syntax, typing rules, and reduction rules for arithmetic and boolean expressions.

Arith and Bool boxes in Figure 1.1. A lambda feature can be defined as a similar collection of BNF, typing, and reduction rules and easily combined with the definitions for arithmetic expressions. Ideally a similar approach could be applied to metatheory proofs, with partial proofs for each features definitions being combined to build complete proofs for a language. Unfortunately, simply combining the text of informal proofs provides no guarantees on the result's correctness. Standard practice [Pie02] is to copy and paste together the informal metatheory proofs and then carefully read over the result, patching up any problems. This approach is perhaps the best we can hope for in pen-and-paper formalizations, which rely on human judgement for verification.

A more promising path forward is provided by proof assistants such as Coq [BC04], Agda [Nor07], ACL2 [KM] and Isabelle/HOL [NWP02], which researchers are increasingly using to formalize programming language metatheory. These tools assist in the construction of proofs represented as derivations using the rules of a formal logic. In contrast to informal pen-and-paper proofs, these derivations can be verified mechanically by checking that each step is a valid application of one of these rules. Mechanical proofs hold the promise of more principled reuse since these derivations are formal objects which can be manipulated and the validity of which can be mechanically certified. The authors of the PoplMark challenge [A+05] noted the importance of *reuse of components* in facilitating experimentation and enabling researchers to share language designs when they identified it as a key challenge in mechanizing metatheory.

Unfortunately, despite having powerful modularity constructs such as *modules* [Mac84], *type classes* [WB89, SO08, GZND11] and expressive forms of *dependent types* [CH86, PM93], the cut-paste-patch approach to language extension in pen-and-paper formalizations has carried over to mechanization of programming language metatheory. A key reason for this state of affairs is that the natural decomposition of a language into features cuts across the modularity boundaries of these constructs[2]. A key contribution of this thesis is

[2]Chapter 3 delves deeper into the details of these challenges.

the development of a more principled approach to language extension using the Coq theorem prover as the vehicle for reuse of both definitions and metatheory proofs. Chapter 3 demonstrates that mechanized metatheory proofs can be effectively decomposed into feature modules which can be mechanically verified and then reused in a number of different languages, while Chapters 4 and 5 develop techniques and tools for building these modules.

1.1.2 Feature Modularity in Object-Oriented Languages

The Java implementation of an interpreter for one of these languages[3] will be distributed among three collections of classes — Arith, Bool, and Lambda. Modularizing each feature into these distinct collections of classes allows clients to tailor the language to their needs. Client code which only needs an arithmetic and boolean expression language, for example, can import only Arith, and Bool, while a client which needs a language with arithmetic expressions and lambda functions can simply include Arith and Lambda. Importantly, because each feature is implemented as a distinct collections of Java classes, the implementation of each feature can be typechecked independently and safely reused in both language designs.

Problems arise when adding new features whose implementations cut across classes. A programmer may wish to add a pretty printing feature to the language, which requires extending the Arith, Bool, and Lambda classes with a new printing method, as shown in Figure 1.2. One approach to implementing this extension is to use *inheritance* [Coo89] to create subclasses of Arith, Bool, and Lambda with pretty printing methods. Inheritance is not flexible enough to support multiple extensions, e.g. the addition of a type checker[4], to the same class

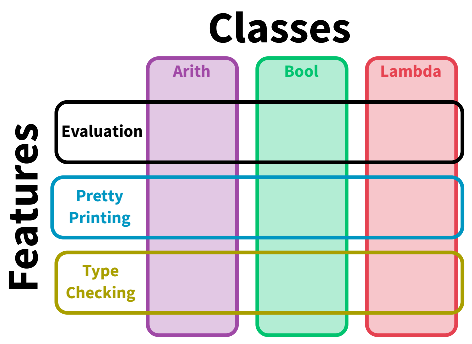


Figure 1.2: Example design of Java implementation of our expression languages.

[3]The implementation of which is guided by the reduction rules in Figure 1.1.
[4]Chapter 2 explores these limitations in more detail.

without duplication, however. The Feature-Oriented
Software Development community has proposed a number of extension mechanisms for overcoming the limitations of inheritance [Bat04, AKL09], but these approaches are purely syntactic — they take the text of an existing program and the text of a refinement and produce the text of a new Java program. Lacking a proper semantics, these refinements cannot be compiled or analyzed in isolation. If a refinement contains a type error, it will not be caught until it is composed with a Java program and the result is analyzed by the Java typechecker. Chapter 2 solves this problem by introducing a type system which allows object-oriented feature modules to be type-checked in isolation.

1.1.3 Efficient Family-Level Reasoning

Different combinations of the five language features mentioned so far (Arith, Bool, Lambda, Pretty Printing, Type Checking) describe different members of a family or *product line* of related expression languages. A key challenge in product-line engineering is efficiently verifying that the realization of every valid product specification is correct, with the notion of correctness depending on the implementation domain. Using our expression languages as an example, each Java program should be free of type-errors, while each formalization should have a complete and correct proof of type soundness.

It is possible to verify a product-line by generating each individual product and checking that it satisfies the desired correctness property. This approach does not scale (even for correctness properties which can be established through automated analyses like type-checking), as the size of the family of languages grows exponentially in the number of available features. Developing efficient analyses is a major focus of the product-line engineering community [ARW+13]. Chapter 6 discusses how modular feature analyses can be efficiently lifted in a principled and provably correct manner, enabling the correctness of the family to be verified without enumerating and checking every product in the line individually.

1.1.4 Summary of Contributions

To summarize the contributions of this dissertation:

1. A more principled approach to sound language extension relying on the Coq proof assistant. The metatheory proofs for a feature's definitions can be independently developed and mechanically checked. These proofs are combined to synthesize proofs of soundness for extended languages.

2. Novel reasoning techniques for extending languages with new values and effects, including a solution for a decades-old problem with reasoning over church-encoded datatypes [PM93].

3. A method for scaling analyses to product-lines expressed as compositions of features. As an example, a constraint-based type sytem for a calculus of object-oriented feature modules with mixins is presented. This type system checks these modules in isolation by generating a set of constraints. The interfaces formed by these constraints are exploited to statically verify that all valid feature combinations are free of type errors.

1.2 Preliminaries and Background

We begin by establishing some basic notions which will be used throughout the thesis.

1.2.1 Modules

The New Oxford American Dictionary defines a *module* in computer science as "any of a number of distinct but interrelated units from which a program may be built up or into which a complex activity may be analyzed." This definition exposes the dual use of modules as both a mechanism for reuse and as a means for understanding complex systems. This thesis explores both definitions of modularity in the context of mechanized reasoning by showing how modules can be leveraged to reuse mechanized verification of meta-theory of programming language extensions and how their abstractions can be exploited to efficiently analyze large families of related artifacts.

The precise definition of a module depends on the domain — every programming language, for example, has a set of modularity constructs which dictate how programs can be decomposed. Broadly speaking, a module is a collection of definitions that it abstracts using an *export* interface. These definitions may reference external definitions which the module abstracts through an *import* interface. Taken together, these interfaces provide an abstraction of the module: whenever the references in a module m are linked to definitions which satisfy m's import interface, its definitions will satisfy the export interface. This abstraction is typically enforced statically through a judgement (m OK) — a type system, for example, checks that a module satisfies an interface specified as types.

Mixin modules [BC90, BL92] are modules equipped with a binary composition operator, $\cdot$, that builds a new module by combining two modules together. This module is built by using the definitions exported by each module to resolve the imports dependencies of the other:

$$\begin{aligned}\mathsf{imports}(m \cdot n) &= \mathsf{imports}(m) \cup \mathsf{imports}(n) \quad - \quad (\mathsf{exports}(m) \cup \mathsf{exports}(n)) \\ \mathsf{exports}(m \cdot n) &= \mathsf{exports}(m) \cup \mathsf{exports}(n)\end{aligned}$$

Ideally this composition operator will preserve well-formedness:

$$\frac{m\ \mathsf{OK} \qquad n\ \mathsf{OK}}{m \cdot n\ \mathsf{OK}}$$

This property allows for compositional reasoning: properties of a composite module can be inferred from properties of its constituent modules. Thus, a complex system built as a composition of modules can be reasoned about by only considering the modules it is built from (each of which can be reasoned about in isolation).

The key to reuse of modules is *abstraction* through their import interfaces. It is through abstraction over the imports that modules can be reused in different contexts and variability can be introduced. In the parlance of product-line design, these imports are *Variation Points* (VP), a standard concept in product line designs [Bas87]. There is a tradeoff here — too much abstraction introduces too much variability, making it hard to develop much reusable infrastructure. It is unreasonable to reuse proofs about the natural numbers when reasoning about Taylor-series, for example. On the other hand, too little abstraction limits reuse of the module. Thus, it is important to identify domains which support abstraction that maximizes both proof construction and variability. There are dual concerns here: identifying abstraction mechanisms for modularization, and then effectively deploying those mechanisms to build modules.

1.2.2 Feature-Oriented Design

Complex systems are often understood as collections of distinguishing properties or features. Our first expression language, for example, can be thought of as a combination of arithmetic and boolean expressions:

$$\mathsf{ABExp} = \mathsf{Arith} + \mathsf{Bool}$$

Alternatively, the second expression language, ALExp, is a combination of arithmetic expressions and lambda abstractions:

$$\mathsf{ALExp} = \mathsf{Arith} + \mathsf{Lambda}$$

These specifications of ABExp and ALExp are expressions in the algebra of features [BKH11]. The domain of this algebra is a set of feature names equipped with the composition operation +. Feature names have no specific meaning (although they often connote an intuitive one). Feature names are given a precise denotation through mappings to concrete representations, with the meaning of + depending on the codomain of a given mapping. The programmer from Section 1.1.2 built the Java mapping constructing the

Java implementation of a language, while the language researcher from Section 1.1.1 gave the Formal mapping generating the syntax, type system, and operational semantics of a language. The simplest form of these functions is as one-to-one mappings from algebraic specifications to their implementations:

$$\mathsf{Java}(\mathsf{ABExp}) = \mathsf{ABExp}_{\mathsf{Java}}$$
$$\mathsf{Formal}(\mathsf{ABExp}) = \mathsf{ABExp}_{\mathsf{Formal}}$$

This naive approach requires manual development of distinct implementations for each possible algebraic specification, which limits reuse between the different variants. While ABExp and ALExp both include arithmetic expressions, this Java mapping recreates the classes of Figure 1.2 for both languages:

$$\mathsf{Java}(\mathsf{ALExp}) = \mathsf{ALExp}_{\mathsf{Java}}$$

In addition to the inefficiencies introduced by maintaining multiple copies of these classes, this approach hinders understanding of the languages by obscuring the commonalities between the two variants. To expose these commonalities and maximize reuse, Java should ideally be implemented as a homomorphism that distributes over the feature algebra's composition operation (+):

$$\begin{aligned}
&\mathsf{Java}(\mathsf{ABExp}) \\
= \quad & \mathsf{Java}(\mathsf{Arith} + \mathsf{Bool}) \\
= \quad & \mathsf{Java}(\mathsf{Arith}) +_{\mathsf{Java}} \mathsf{Java}(\mathsf{Bool}) \\
= \quad & \mathsf{Arith}_{\mathsf{Java}} \ +_{\mathsf{Java}} \ \mathsf{Bool}_{\mathsf{Java}}
\end{aligned}$$

This allows $\mathsf{Java}(\mathsf{ALExp})$ to reuse the $\mathsf{Arith}_{\mathsf{Java}}$ module:

$$\mathsf{Java}(\mathsf{ALExp}) = \mathsf{Arith}_{\mathsf{Java}} +_{\mathsf{Java}} \mathsf{Lambda}_{\mathsf{Java}}$$

The essence of feature modularity for a given feature set $\mathcal{F}$ is a homomorphism δ whose range includes a module F_δ implementing each feature $\mathsf{F} \in \mathcal{F}$ and whose codomain is equipped with a composition operator $+_\delta$ which can compose these modules. The key challenge to achieving feature modularity is that features can cut across modularity boundaries of a codomain [LHBL06], preventing it from being a homomorphism.

1.2.3 Semantic Feature Modularity

In addition to being able to decompose a system into reusable feature modules, we also want to be able to derive properties of a composition of feature modules from properties of its constituent features. The modularization of properties of feature modules is just another application of the above ideas, in that it reduces (or decomposes) *reasoning* about a composition of modules to reasoning about individual features.

A property P holds for object e if there exists some evidence of this fact — in this thesis, this evidence is a formal mathematical *proof* or derivation in some well-defined system. We denote a proof r of property $P(e)$ as $\vdash r \; : \; P(e)$. Semantic feature modularity of a property π for a mapping δ is realized as a mapping to a property and a proof ρ of π for each element of the range of δ:

$$\frac{\vdash \rho(\mathsf{A}) \; : \; \pi(\delta(\mathsf{A})) \qquad \vdash \rho(\mathsf{B}) \; : \; \pi(\delta(\mathsf{B}))}{\vdash \rho(\mathsf{A} + \mathsf{B}) \; : \; \pi(\delta(\mathsf{A} + \mathsf{B}))}$$

As an example, we may wish to verify that the Java implementation of ABExp doesn't have any type errors:

$$\mathsf{WF}_\pi(\mathsf{ABExp}) = \mathsf{Java}(\mathsf{ABExp}) \; \textbf{OK}$$

One approach to establishing $\mathsf{WF}_\pi(\mathsf{ABExp})$ is to generate $\mathsf{Java}(\mathsf{ABExp})$ and analyze it with Java's typechecker. To avoid repeated clock cycles when typing both ABExp and ALExp, Chapter 2 introduces a technique for checking the implementation of a feature in isolation. It furthermore shows how to combine the proofs of WF_π, allowing a composition of features to be typed by using the proofs of WF_π for individual features:

$$\frac{\vdash \mathsf{WF}_\rho(\mathsf{Arith}) \;:\; \mathsf{WF}_\pi(\mathsf{Arith}) \qquad \vdash \mathsf{WF}_\rho(\mathsf{Bool}) \;:\; \mathsf{WF}_\pi(\mathsf{Bool})}{\vdash \mathsf{WF}_\rho(\mathsf{ABExp}) \;:\; \mathsf{WF}_\pi(\mathsf{ABExp})}$$

$$(= \;\; \vdash \mathsf{WF}_\rho(\mathsf{Arith} + \mathsf{Bool}) \;:\; \mathsf{WF}_\pi(\mathsf{Arith} + \mathsf{Bool}))$$

Thus, feature modularity can be achieved for the semantic mapping WF_ρ, just as for Java and Formal. For this set of codomains, we can completely modularize a feature F as a tuple of the *syntactic* mappings, i.e. the Java implementation of F, and the *semantic* mappings to evidence of properties of the syntactic mappings, i.e. that $\mathsf{F}_{\mathsf{Java}}$ is free of any type errors:

$$\mathsf{F} = [\mathsf{F}_{\mathsf{Java}}, \mathsf{F}_{\mathsf{WF}_\rho}]$$

Feature composition becomes tuple composition with specialized composition operators for each component:

$$\begin{aligned} &\mathsf{F} + \mathsf{G} \\ &\quad = \; [\mathsf{F}_{\mathsf{Java}}, \mathsf{F}_{\mathsf{WF}_\rho}] + [\mathsf{G}_{\mathsf{Java}}, \mathsf{G}_{\mathsf{WF}_\rho}] \\ &\quad = \; [\mathsf{F}_{\mathsf{Java}} +_{\mathsf{Java}} \mathsf{G}_{\mathsf{Java}},\; \mathsf{F}_{\mathsf{WF}_\rho} +_\rho \mathsf{G}_{\mathsf{WF}_\rho}] \end{aligned}$$

Feature modularity for semantic mappings has two important benefits:

1. It allows feature modules to be analyzed and understood in isolation, and

2. Once a property has been established for a feature, multiple products can reuse the results of a (potentially expensive) analysis. The proofs of $\mathsf{WF}_\pi(\mathsf{ABExp})$ and $\mathsf{WF}_\pi(\mathsf{ALExp})$ can both reuse the proofs of $\mathsf{WF}_\pi(\mathsf{Arith})$, for example.

Semantic module composition is monotonic: what was true before modules were composed remains valid after composition, although the scope of validity may be qualified. This approach is standard in feature-based designs [BB08]. Typical SPL tools support syntactic modules *without* semantic modularity, a key impediment to scaling SPL analyses.

1.2.4 Feature Interactions

For clarity, the previous discussion used the + operator to compose features, but products are usually built using the binary × operator that integrates feature interactions [BKH11] into composition. The product of two features F and G includes an extra feature, $\mathsf{F\#G}$, in their composition:

$$\mathsf{F} \times \mathsf{G} = \mathsf{F\#G} + \mathsf{F} + \mathsf{G} \tag{1.1}$$

$\mathsf{F\#G}$ denotes the interaction of features F and G. F and G are presumed to work correctly in isolation, but may need some coordination (modifications) to work correctly together. $\mathsf{F\#G}$ contains the necessary modifications to F and G.

The classical example of feature interactions is the problem of fire and flood control [Kan05]. Let b denote the design of a building. The flood control feature adds water sensors to every floor of b. If standing water is detected, the water main to b is turned off. The fire control feature adds fire sensors to every floor of b. If fire is detected, sprinklers are turned on. Adding flood or fire control to the building (e.g. $\mathsf{flood} + \mathsf{b}$ and $\mathsf{fire} + \mathsf{b}$) is straightforward. However, adding both ($\mathsf{flood} + \mathsf{fire} + \mathsf{b}$) is problematic: if fire is detected, the sprinklers turn on, standing water is detected, the water main is turned off, and the building burns down. This is not the intended semantics of the composition of the flood, fire, and b features. The fix is to apply an additional extension, labeled $\mathsf{flood\#fire}$, which is the interaction of flood and fire. $\mathsf{flood\#fire}$ represents the changes (extensions) that are needed to make the flood and fire features work correctly together. The correct building design is $\mathsf{flood\#fire} + \mathsf{flood} + \mathsf{fire} + \mathsf{b}$.

As × reduces to + and interaction features are themselves features, the previous discussion also applies to products built with this new operator[5]. $\rho(\mathsf{F}\#\mathsf{G})$ for a semantic mapping ρ is a semantic module that is only required when both the F and G features are composed together. These modules typically export proofs about the exports of F assumed by G and vice versa. From a semantic viewpoint, these are the only interactions that matter.[6]

An $\times$-product of n features results in $O(2^n)$ interactions (i.e. all possible feature combinations). Fortunately, the *vast* majority of feature interactions are empty, meaning that they correspond to the identity transformation $\mathsf{0}$, whose properties are:

$$\mathsf{0} + \mathsf{f} \;=\; \mathsf{f} + \mathsf{0} \;=\; \mathsf{f} \tag{1.2}$$

Most non-empty interactions are pairwise (2-way). Only very rarely do higher-order interactions arise. As an example, consider the $\times$-product of A, B, and C, where all 2- and 3-way interactions except $\mathsf{A}\#\mathsf{B}$ equal $\mathsf{0}$:

$$\begin{aligned}
&\mathsf{A} \times \mathsf{B} \times \mathsf{C} \\
&\quad = \mathsf{A}\#\mathsf{B}\#\mathsf{C} + \mathsf{A}\#\mathsf{B} \\
&\qquad + \mathsf{A}\#\mathsf{C} + \mathsf{A} + \mathsf{B}\#\mathsf{C} \\
&\qquad + \mathsf{B} + \mathsf{C} \\
&\quad = \mathsf{A}\#\mathsf{B} + \mathsf{A} + \mathsf{B} + \mathsf{C}
\end{aligned}$$

1.2.5 Feature Models

Not all compositions of features are meaningful: some features require the presence or absence of other features. A car's stereo system, for example, requires an antenna. Feature models define the compositions of features that produce meaningful languages. A *feature model* is a context sensitive grammar, consisting of a context free grammar whose sentences

[5]Although we do not deal with feature replication in this thesis, the way feature replicas are handled in an axiomatization is by making each replica a distinct feature. So if feature A is replicated twice, its replicas are denoted by unique features A_1 and A_2.

[6]Assuming that the correctness of a product is completely specified by the properties that are being proved

define a superset of all legal feature expressions, and a set of constraints (the context sensitive part) that eliminates nonsensical sentences [Bat05a]. The grammar of feature model P (below) defines eight sentences (features k, i, j are optional; b is mandatory). Its constraints limit the legal sentences to those that have at least one optional feature, and also require that feature j must be included if k is selected.

$$\begin{array}{lcrl} \mathsf{P} & : & [\mathsf{k}]\ [\mathsf{i}]\ [\mathsf{j}]\ \mathsf{b}; & \text{// context free grammar} \\ & & \mathsf{k}\ \vee\ \mathsf{j}\ \vee\ \mathsf{i}; & \text{// additional constraints} \\ & & \mathsf{k} \Rightarrow \mathsf{j}; & \end{array}$$

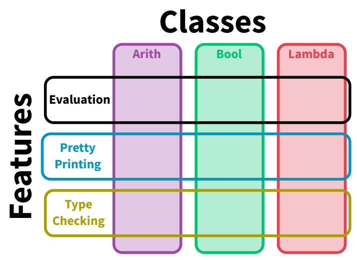
A sentence of a feature model ('kjb') corresponds to the $\times$-product of its features in the algebra of features ($\mathsf{k} \times \mathsf{j} \times \mathsf{b}$). The set of sentences of a feature model specifies all the valid members of a product line. Having covered the basic theory of features needed for this dissertation, we now continue to our first application of semantic feature modularity: type-checking feature modules for object-oriented software product lines.

Chapter 2

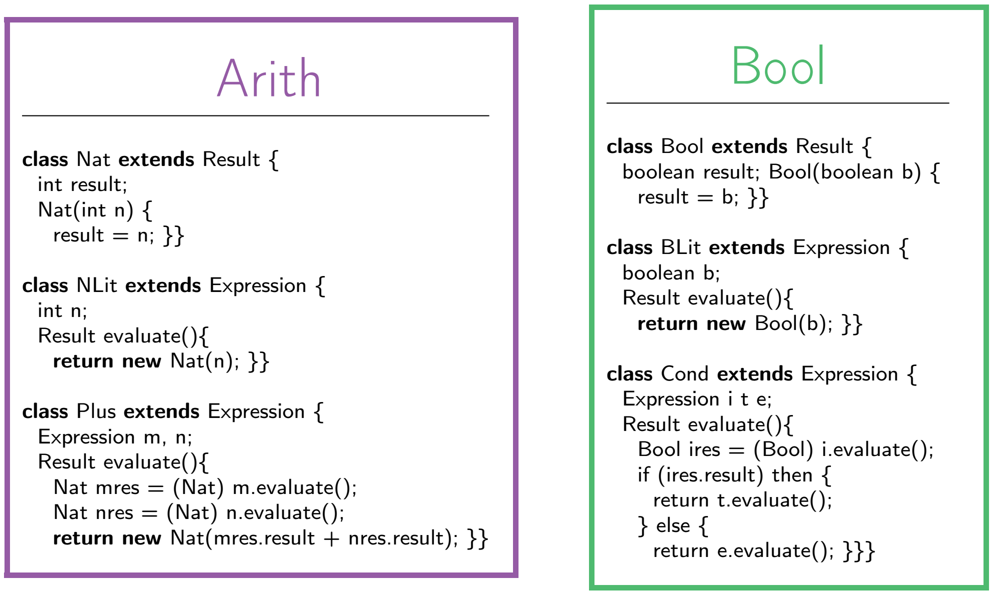
Feature Modularity in Software

2.1 Introduction

The development of almost every design begins with specifying the high-level distinguishing features. In software design, only after these features have been identified through *requirements analysis* do developers begin to map these features onto a concrete program. Most programming languages include constructs for structuring software into modular components in such a way that programs are easier to understand, reuse, and evolve. Mapping each feature to a module in the target language lifts these benefits to the design level, enabling reuse and extension of the *features* themselves. When features cut across the modularity boundaries of the target language, the connection between design and implementation is obscured, and these benefits are lost. The challenge of feature modularity in software design is to equip programming languages with modularity constructs which make it possible to establish a homomorphic mapping from features to program modules.



In object-oriented languages, the main modularity constructs are objects and classes. Object-oriented programs typically consist of multiple objects communicating with each other. The code needed to coordinate the object interactions for a given feature often needs to be distributed across a number of different objects[1], cross-cutting the modularity boundaries of object-oriented languages. As an example of this problem, consider the family of expression languages from the introduction, shown again in the figure to the right. The collections of classes in Arith, Bool, and Lambda each implement the basic expressions and evaluation functions. The Pretty Printing and Type Checking features implement printing and type checking methods in each class. Figure 2.1 presents an example implementation of the basic classes implementing the Arith and Bool features.



```
class Nat extends Result {
  int result;
  Nat(int n) {
    result = n; }}

class NLit extends Expression {
  int n;
  Result evaluate(){
    return new Nat(n); }}

class Plus extends Expression {
  Expression m, n;
  Result evaluate(){
    Nat mres = (Nat) m.evaluate();
    Nat nres = (Nat) n.evaluate();
    return new Nat(mres.result + nres.result); }}
```

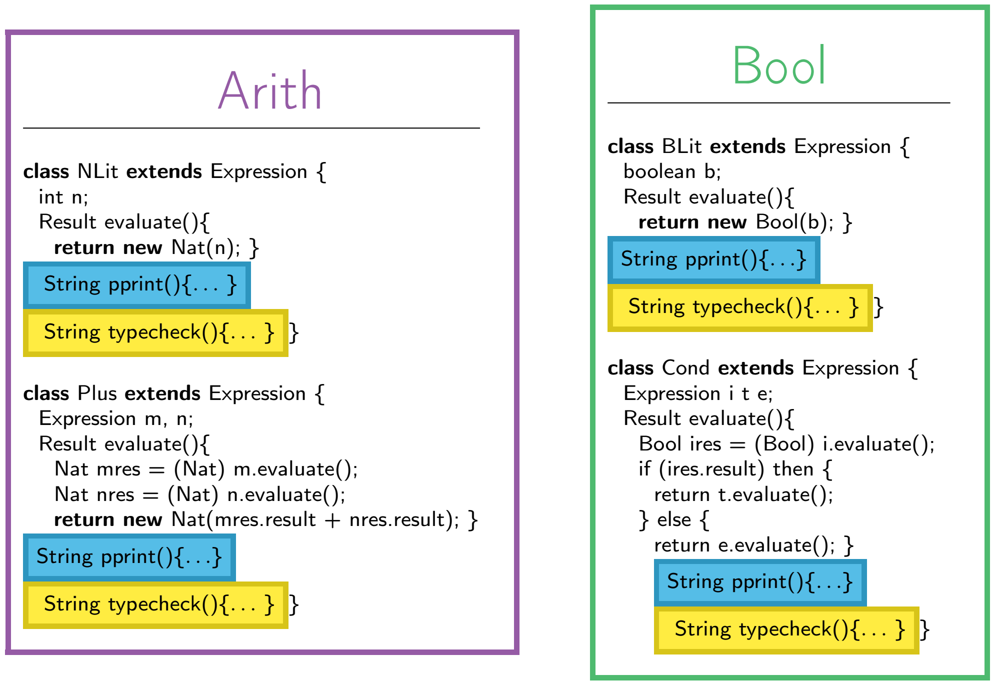
```
class Bool extends Result {
  boolean result; Bool(boolean b) {
    result = b; }}

class BLit extends Expression {
  boolean b;
  Result evaluate(){
    return new Bool(b); }}

class Cond extends Expression {
  Expression i t e;
  Result evaluate(){
    Bool ires = (Bool) i.evaluate();
    if (ires.result) then {
      return t.evaluate();
    } else {
      return e.evaluate(); }}}
```

Figure 2.1: Java implementation of a simple arithmetic and boolean expression language.

The feature-oriented software design community has developed a number of approaches to syntactic feature modularity, most of which can be broadly classified as either *projectional* or *compositional*. Projectional approaches are often used in legacy systems [FKA+13] and encode all variations in a single meta-module in which each region of

[1]The part of an object that implements a feature is known as a *role* in collaboration-based design.

code is associated with a specific feature or combination of features, a process also known as *coloring* [KAK08]. This is effectively a simplified version of SysGen [C28]. A specific class is generated by projecting out the code associated with a set of selected features. Under this model, the NLit, Plus, BLit, and Cond classes would contain marked pieces of code for the Pretty Printing and TypeChecking features, as shown in Figure 2.2.



```
class NLit extends Expression {
  int n;
  Result evaluate(){
    return new Nat(n); }
String pprint(){. . . }
String typecheck(){. . . } }

class Plus extends Expression {
  Expression m, n;
  Result evaluate(){
    Nat mres = (Nat) m.evaluate();
    Nat nres = (Nat) n.evaluate();
    return new Nat(mres.result + nres.result); }
String pprint(){. . .}
String typecheck(){. . . } }
```

```
class BLit extends Expression {
  boolean b;
  Result evaluate(){
    return new Bool(b); }
String pprint(){. . .}
String typecheck(){. . . } }

class Cond extends Expression {
  Expression i t e;
  Result evaluate(){
    Bool ires = (Bool) i.evaluate();
    if (ires.result) then {
      return t.evaluate();
    } else {
      return e.evaluate(); }
      String pprint(){. . .}
      String typecheck(){. . . } }
```

Figure 2.2: Java implementation of expression language variants using coloring.

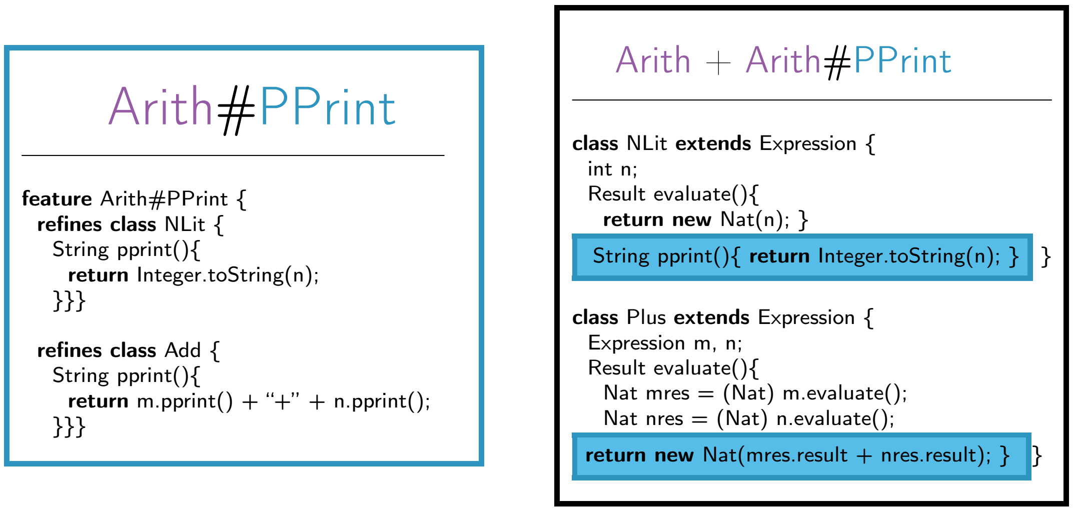
Compositional approaches, on the other hand, take a more traditional approach to feature modularity by expressing features as collections of extensions implementing a feature's functionality. Composition tools such as AHEAD [Bat04] and FeatureHouse [AKL09] define a meta-language with constructs for extensions to modules in the base language and grouping these extensions. A composition operator synthesizes these components into a program in the base language.

The challenge for implementing this operator for object-oriented languages is that their standard extension mechanism of *inheritance* [Coo89] is not flexible enough to support every extension a feature may require. While a superclass can be defined without specifying the possible subclasses (extensions), a subclass is defined with respect to a specific superclass. To see why this is a problem, consider the extensions required by the Type Checking feature. Both the Pretty Printing and Type Checking features extend the BLit and Plus classes. In order for the extensions needed to implement Type Checking to be expressed using inheritance, they must be expressed as subclasses of these two classes. If Pretty Printing has already subclassed BLit and Plus, however, Type Checking needs to extend those subclasses instead. Thus, Type Checking needs to be implemented as two different sets of classes, one for each of the possible combinations of feature selections it can be applied to. In the worst case, the size of this set of modules is exponential in the number of features, resulting in the so-called *library scalability problem* [BSST93, Big94]. The standard solution to this problem is for the meta-language to include a construct for subclasses with abstract superclasses or *mixins*[2].

The AHEAD tool suite [Bat04] includes the Jak language for expressing features as collections of Java class definitions and *refinements* (mixins). A class refinement is a modification to an existing class which adds new fields, new methods, and wraps existing methods. Composing a feature component with a program introduces new classes to the program and applies the refinements of the feature to the program's existing classes. Figure 2.3 has an example of such a Jak module implementing pretty printing for arithmetic expressions. The + operator generates Java programs by using Jak feature modules to refine Java programs. Figure 2.3 also shows an example of a refinement of the Arith class, with the added methods highlighted (the Expression class is updated similarly). Jak components are purely syntactic— the semantics of feature modules are defined by a reduction to a base language. While specific products can be typed using the type system of the target language, the components cannot. In addition to limiting analysis of individual features,

[2]This original definition of mixin for objects was given by Bracha and Cook [BC90] and later generalized to the generic definition for modules given in Section 1.2.1. Note that the former is a specialization of the latter.

this is an important barrier to type-checking an entire feature-oriented software product line, as each product must be generated and type-checked. The remainder of this chapter addresses how to type-check these Jak-style mixin modules, while Chapter 6 shows how to leverage the resulting modules to lift typechecking to the product-line level.



```
feature Arith#PPrint {
  refines class NLit {
    String pprint(){
      return Integer.toString(n);
    }}}

  refines class Add {
    String pprint(){
      return m.pprint() + "+" + n.pprint();
    }}}
```

```
class NLit extends Expression {
  int n;
  Result evaluate(){
    return new Nat(n); }
String pprint(){ return Integer.toString(n); } }

class Plus extends Expression {
  Expression m, n;
  Result evaluate(){
    Nat mres = (Nat) m.evaluate();
    Nat nres = (Nat) n.evaluate();
return new Nat(mres.result + nres.result); } }
```

Figure 2.3: Pretty printing extensions to the Arith class from Figure 2.1 using Jak mixins.

We formalize our compositional approach to feature-based software development using an object-oriented kernel language extended with features, called *Lightweight Feature Java* (LFJ). Composition of LFJ feature modules generates a program in *Lightweight Java* [SSP07], a subset of Java that includes a formalization in the Coq proof assistant [BC04], using the Ott tool [SNO$^+$07]. The design of LFJ is inspired by the Jak language of the AHEAD tool suite, with LJ replacing Java as the base language. A program in LFJ is a set of features containing classes and class refinements. Multiple products can be constructed by selecting and composing appropriate features according to a *product specification*– a composition of features. Feature modules in LFJ have interfaces that govern their composition. In lieu of using explicit feature interfaces to type LFJ feature modules, we instead infer the necessary feature interfaces from the constraints generated by a constraint-based type system for LFJ. The type system and its safety are formalized in Coq.

2.2 Lightweight Feature Java

Lightweight Feature Java (LFJ) is a kernel language that captures the key concepts of feature-based product lines of Java programs. LFJ is based on Lightweight Java (LJ), a minimal imperative subset of Java [SSP07]. LJ supports classes, mutable fields, constructors, single inheritance, methods and dynamic method dispatch. LJ does not include local variables, field hiding, interfaces, inner classes, or generics. Appendix A presents the complete syntax, type system and operational semantics for LJ. This imperative kernel provides a minimal foundation for studying a type system for feature-oriented programming. LJ is more appropriate for this work than Featherweight Java [Pie02] because of its treatment of constructors. When composing features, it is important to be able to add new member variables to a class during refinement. Featherweight Java requires all member variables to be initialized in a single constructor call. As a result, adding a new member variable causes all previous constructor calls to be invalid. Lightweight Java allows such refinements through its support of more flexible initialization of member variables. In addition, Lightweight Java has a full formalization in Coq, which we extended to prove the soundness of LFJ mechanically.

The syntax LFJ add to LJ in order to support feature-oriented programming is given in Figure 2.4. A feature definition FD maps a feature name F to a list of class declarations $\overline{\mathsf{cld}}$ and a list of class refinements $\overline{\mathsf{rcld}}$. A class refinement rcld includes a class name dcl, a set of LJ field and method introductions, $\overline{\mathsf{fd}}$ and $\overline{\mathsf{md}}$, a set of method refinements $\overline{\mathsf{rmd}}$, and the name of the updated parent class cl. A method refinement advises a method with signature ms with two lists of LJ statements $\overline{\mathsf{s}}$ and an updated return value y. When applied to an existing method, a method refinement wraps the existing method body with the advice. The parameters of the original method are passed implicitly because the refinement has the same signature as the method it refines. The feature table FT contains the set of features included in a product line. A product specification PS selects a distinct list of feature names from the feature table.

Feature Table	Class refinement
$\mathsf{FT} ::= \{\overline{\mathsf{FD}}\}$	$\mathsf{rcld} ::= \textbf{refines class } \mathsf{dcl} \textbf{ extending } \mathsf{cl}\{\overline{\mathsf{fd}}; \overline{\mathsf{md}}; \overline{\mathsf{rmd}}\}$
Product specification	Method refinement
$\mathsf{PS} ::= \overline{\mathsf{F}}$	$\mathsf{rmd} ::= \textbf{refines method } \mathsf{ms}\ \{\mathsf{rmb}\}$
Feature declarations	Method refinement
$\mathsf{FD} ::= \textbf{feature } \mathsf{F}\ \{\overline{\mathsf{cld}}; \overline{\mathsf{rcld}}\}$	$\mathsf{rmb} ::= \bar{\mathsf{s}}; \textbf{Super()}; \bar{\mathsf{s}}; \textbf{return } \mathsf{y}$

Figure 2.4: Modified Syntax of Lightweight Feature Java.

2.2.1 Feature Composition

A LJ program can be modelled as a partial function from class names to their definitions: $\mathsf{CT} : \mathsf{dcl} \rightarrow \mathsf{cld}$. In the operational semantics of LJ, this function is concretely realized as the function $\textbf{path} : \mathsf{P} \rightarrow \mathsf{dcl} \rightarrow \mathsf{cld}$ which looks up a class definition in a given program. In this context, CT is simply the path specialized on P: $\mathsf{CT} = \textbf{path}_{\mathsf{P}}$. The composition operator builds a new LJ program by refining the definitions of an existing LJ program: $+ \ : \ \mathsf{FD} \rightarrow \mathsf{P} \rightarrow \mathsf{P}$. The semantics of composition are described by the new class table produced when a feature $\textbf{feature } \mathsf{F}\ \{\overline{\mathsf{cld}}; \overline{\mathsf{rcld}}\}$ is composed with an LJ program P produces a new mapping:

$$\mathsf{CT}'(\mathsf{dcl}) = \textbf{path}_{\textbf{feature } \mathsf{F}\{\overline{\mathsf{cld}};\overline{\mathsf{rcld}}\}+\mathsf{P}} \begin{cases} \textbf{path}_{\overline{\mathsf{cld}}}(\mathsf{dcl}) & \mathsf{dcl} \in \overline{\mathsf{cld}} \\ \overline{\mathsf{rcld}} \cdot \mathsf{cld} & \mathsf{dcl} \notin \overline{\mathsf{cld}} \wedge \textbf{path}_{\mathsf{P}}(\mathsf{dcl}) = \mathsf{cld} \end{cases}$$

In the case that F introduces a class named dcl, CT' returns this class, ignoring any previous declarations and refinements. Otherwise, CT' finds the definition of dcl in the previous program using the original $\mathsf{CT} = \textbf{path}_{\mathsf{P}}$ function and returns the resulting class definition, cld, refined by $\overline{\mathsf{rcld}}$. If a class refinement rcld in $\overline{\mathsf{rcld}}$ is named dcl, the $\cdot$ operator builds a refined class by first advising the methods of cld with the method refinements in rcld. The fields and methods introduced by rcld are then added to this class and its parent is set to the superclass named in rcld. CT' is undefined if cld lacks a method refined by rcld.

A product specification builds an LJ program by recursively composing the features it names in this manner, starting with the empty LJ program. Each LFJ feature table can construct a family of programs through composition, with the set of class definitions determined by the sequence of features which produced them. The class hierarchy is also potentially different in each program: refinements can alter the parent of a class, and two mutually exclusive features can introduce a class with the same name but with different parents.

2.2.2 Feature Modularity in LFJ

The mixin-based features presented here do not meet the definition of modules presented in the previous chapter because they allow classes to be overwritten, removing fields and methods exported by that class. This is due to the original motivation of LFJ as a formalization of the mixins used in the AHEAD [Bat04] tool suite. AHEAD mixins implement features as refinements of existing definitions, with composition taking a refinement and an existing Java program and producing a new Java program. This approach is modelled by the semantics of LFJ as a reduction to LJ. With a few modification to the + operator, however, LFJ features can form the foundation for a proper module system. The first modification is to restrict + to fail when two modules exporting the same definition are composed. The other modification is to extend the operator to compose class and method refinements, as its current semantics assume that a refinement is always applied to an existing definition.

2.3 LFJ Type System

The constraint-based type system for LFJ is based on a constraint-based type system for LJ. Both retain the premises that depend on the structure of the construct being typed and convert those that rely on external information into constraints. By using constraints, the external typing requirements for each feature are made explicit, separating derivation of these requirements from their satisfaction. Generating a set of constraints for a feature is separated from consideration of which product specifications have a combination of features satisfying these constraints.

The constraints used to type LFJ are given in Figure 2.5 and are divided into four categories. The two composition constraints guarantee successful composition of a feature F by requiring that refined classes and methods be introduced by a feature in a product line before F. The two uniqueness constraints ensure that member names are not overloaded within the same class, a restriction in the LJ formalization. The structural constraints come from the standard LJ type system and determine the members of a class and its inheritance hierarchy in the final program. The subtype constraint is particularly important because the class hierarchy is malleable until composition; if it were static, constraints that depend on subtyping could be reduced to other constraints or eliminated entirely. The feature constraint specifies that if a feature F is included in a product specification its constraints must be satisfied.

Composition Constraints

dcl $\textbf{introduces}$ ms $\textbf{before}$ F

dcl $\textbf{introduced before}$ F

Uniqueness Constraints

$\mathsf{cl}\ \mathsf{f}$ $\textbf{unique in}$ dcl

$\mathsf{cl}\ \mathsf{m}\ (\overline{\mathsf{vd}_\mathsf{k}}^{\mathsf{k}})$ $\textbf{unique in}$ dcl

Feature Constraint

$\mathbf{In}_\mathsf{F} \Rightarrow \overline{\xi_\mathsf{k}}^{\mathsf{k}}$

Structural Constraints

$\mathsf{cl}_1 \prec \mathsf{cl}_2$

$\mathsf{cl}_2 \prec \mathbf{ftype}(\mathsf{cl}_1, \mathsf{f})$

$\mathbf{ftype}(\mathsf{cl}_1, \mathsf{f}) \prec \mathsf{cl}_2$

$\mathbf{mtype}(\mathsf{cl}, \mathsf{m}) \prec \overline{\mathsf{cl}_\mathsf{k}}^{\mathsf{k}} \to \mathsf{cl}$

$\mathbf{defined}(\mathsf{cl})$

$\mathsf{f} \notin \mathbf{fields}(\mathbf{parent}(\mathsf{dcl}))$

$\mathbf{pmtype}(\mathsf{dcl}, \mathsf{m}) = \tau$

Figure 2.5: Syntax of Lightweight Feature Java typing constraints.

The typing rules for LFJ are found in Figure 2.6-2.9 and rely on judgements of the form $\vdash \mathsf{J} \mid \xi$, where J is a LFJ typing judgement and ξ is a set of constraints. ξ provides an explicit interface which guarantees that J holds in any product specification that satisfies ξ. Typing rules for statements, methods, and classes are those from LJ augmented with constraints. Typing rules for class and method refinements in a feature F are similar to those for the objects they refine, but require that the refined class or method be introduced in a feature that comes before the F in a product specification. Method refinements do not have to check that the names of their parameters are distinct and that their parameter types

$\boxed{\Gamma \vdash \mathsf{s} \mid \mathcal{C}}$ Statement well-formed in context subject to constraints

$$\frac{\overline{\Gamma \vdash \mathsf{s_k} \mid \mathcal{C}_\mathsf{k}}^{\mathsf{k}}}{\Gamma \vdash \{\mathsf{s_k}\} \mid \bigcup_\mathsf{k} \mathcal{C}_\mathsf{k}} \quad \text{(WF-Block)}$$

$$\frac{\Gamma(\mathsf{x}) = \tau_1 \qquad \Gamma(\mathsf{var}) = \tau_2}{\Gamma \vdash \mathsf{var} = \mathsf{x}; \mid \{\tau_1 \prec \tau_2\}} \quad \text{(WF-Var-Assign)}$$

$$\frac{\Gamma(\mathsf{x}) = \tau_1 \qquad \Gamma(\mathsf{var}) = \tau_2}{\Gamma \vdash \mathsf{var} = \mathsf{x.f}; \mid \{\mathbf{ftype}(\tau_1, \mathsf{f}) \prec \tau_2\}} \quad \text{(WF-Field-Read)}$$

$$\frac{\Gamma(\mathsf{x}) = \tau_1 \qquad \Gamma(\mathsf{y}) = \tau_2}{\Gamma \vdash \mathsf{x.f} = \mathsf{y}; \mid \{\tau_2 \prec \mathbf{ftype}(\tau_1, \mathsf{f})\}} \quad \text{(WF-Field-Write)}$$

$$\frac{\begin{array}{c}\Gamma(\mathsf{x}) = \tau_1 \qquad \Gamma(\mathsf{y}) = \tau_2 \\ \Gamma \vdash \mathsf{s_1} \mid \mathcal{C}_1 \qquad \Gamma \vdash \mathsf{s_2} \mid \mathcal{C}_2 \\ \mathcal{C}_3 = \{\tau_2 \prec \tau_1 \vee \tau_1 \prec \tau_2\}\end{array}}{\Gamma \vdash \mathbf{if}\ \mathsf{x} == \mathsf{y}\ \mathbf{then}\ \mathsf{s_1}\ \mathbf{else}\ \mathsf{s_2} \mid \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3} \quad \text{(WF-If)}$$

$$\frac{\Gamma(\mathsf{var}) = \tau_1 \qquad \mathbf{type}\ (\mathsf{cl}) = \tau_2}{\Gamma \vdash \mathsf{var} = \mathbf{new}\ \mathsf{cl}() \mid \{\tau_2 \prec \tau_1\}} \quad \text{(WF-New)}$$

$$\frac{\begin{array}{c}\Gamma(\mathsf{x}) = \tau \qquad \Gamma(\mathsf{var}) = \pi \qquad \overline{\Gamma(\mathsf{y_k}) = \pi_\mathsf{k}}^{\mathsf{k}} \\ \mathcal{C} = \{\mathbf{mtype}(\tau, \mathsf{meth}) \prec \overline{\pi_\mathsf{k}}^{\mathsf{k}} \to \pi\}\end{array}}{\Gamma \vdash \mathsf{var} = \mathsf{x.meth}(\overline{\mathsf{y_k}}^{\mathsf{k}}) \ \mid \mathcal{C}} \quad \text{(WF-MCall)}$$

Figure 2.6: Typing Rules for LJ and LFJ statements.

and return type are well-formed: a method introduction with these checks must precede the refinement in order for it to be well-formed. Features wrap the constraints on their introductions and refinements in a single feature constraint. The constraints on a feature table are the union of the constraints on each of its features.

$\boxed{\vdash_{\tau,\mathsf{F}} \mathsf{md} \mid \mathcal{C}}$ Method well-formed in class τ and feature F with constraints $\mathcal{C}$

$$\frac{\begin{array}{c}\mathbf{distinct}(\overline{\mathsf{var_k}}^{\mathsf{k}}) \qquad \overline{\mathbf{type}(\mathsf{cl_k}) = \tau_\mathsf{k}}^{\mathsf{k}} \qquad \mathbf{type}(\mathsf{cl}) = \tau' \\ \Gamma = [\overline{\mathsf{var_k} \mapsto \tau_\mathsf{k}}^{\mathsf{k}}][\mathbf{this} \mapsto \tau] \qquad \overline{\Gamma \vdash \mathsf{s}_\ell \mid \mathcal{C}_\ell}^{\ell} \qquad \Gamma(\mathsf{y}) = \tau''\end{array}}{\vdash_\tau \mathsf{cl\ meth}\ (\overline{\mathsf{cl_k\ var_k}}^{\mathsf{k}})\ \{\overline{\mathsf{s}_\ell}^{\ell}\ \mathbf{return}\ \mathsf{y};\} \mid \{\tau'' \prec \tau', \overline{\mathbf{defined}\ \mathsf{cl_k}}^{\mathsf{k}}\} \cup \bigcup_\ell \mathcal{C}_\ell} \quad \text{(WF-Method)}$$

$\boxed{\vdash \mathsf{cld} \mid \mathcal{C}}$ Class well-formed subject to constraints $\mathcal{C}$

$$\frac{\begin{array}{c}\mathbf{distinct}(\overline{\mathsf{f_j}}) \qquad \mathbf{distinct}(\overline{\mathsf{m_k}}) \qquad \mathsf{dcl} \neq \mathsf{cl} \qquad \mathbf{type}(\mathsf{dcl}) = \tau \qquad \overline{\vdash_\tau \mathsf{cl_k\ meth_k}\ (\overline{\mathsf{cl}_{\ell,\mathsf{k}}\ \mathsf{var}_{\ell,\mathsf{k}}}^{\ell})\ \mathsf{mb_k} \mid \mathcal{C}_\mathsf{k}}^{\mathsf{k}} \\ \xi = \bigcup_\mathsf{j}\{\mathsf{f_j} \notin \mathbf{fields}(\mathbf{parent}(\mathsf{dcl}))\} \qquad \upsilon = \bigcup_\mathsf{j}\{\mathsf{cl_j\ f_j}\ \mathbf{unique\ in}\ \mathsf{dcl}\} \qquad \upsilon' = \bigcup_\mathsf{k}\{\mathsf{cl_k\ meth_k}\ (\overline{\mathsf{cl}_{\ell,\mathsf{k}}\ \mathsf{var}_{\ell,\mathsf{k}}}^{\ell})\ \mathbf{unique\ in}\ \mathsf{dcl}\} \\ \xi' = \bigcup_\mathsf{k}\{\mathbf{pmtype}(\mathsf{dcl}, \mathsf{meth_k}) = \overline{\mathsf{cl}_{\ell,\mathsf{k}}}^{\ell} \to \mathsf{cl_k}\}\end{array}}{\vdash \mathbf{class}\ \mathsf{dcl}\ \mathbf{extends}\ \mathsf{cl}\ \{\overline{\mathsf{cl_j\ f_j}}^{\mathsf{j}};\ \overline{\mathsf{cl_k\ meth_k}\ (\overline{\mathsf{cl}_{\ell,\mathsf{k}}\ \mathsf{var}_{\ell,\mathsf{k}}}^{\ell,\mathsf{k}})\ \mathsf{mb_k}}^{\mathsf{k}}\ \} \mid \bigcup_\mathsf{k} \mathcal{C}_\mathsf{k} \cup \{\mathbf{defined}\ \mathsf{cl}, \overline{\mathbf{defined}\ \mathsf{cl_j}}^{\mathsf{j}}\} \cup \xi \cup \xi' \cup \upsilon \cup \upsilon'} \quad \text{(WF-Class)}$$

Figure 2.7: Typing Rules for LJ methods and classes.

Once the constraints $\mathcal{C}$ for a feature table are generated according to the rules in Figure 2.9, we can check whether a specific product specification PS satisfies $\mathcal{C}$ using the rules in Figure 2.10. Feature constraints for a feature F are satisfied when F is not included

$\boxed{\vdash_{\tau,\mathsf{F}} \mathsf{rmd} \mid \mathcal{C}}$ Refined method well-formed in class τ and feature F subject to constraints $\mathcal{C}$

$$\frac{\begin{array}{c}\textbf{type}(\mathsf{cl}) = \tau' \qquad \Gamma = [\overline{\mathsf{var_k} \mapsto \tau_\mathsf{k}}^{\mathsf{k}}][\textbf{this} \mapsto \tau] \\ \Gamma(\mathsf{y}) = \tau'' \qquad \overline{\Gamma \vdash \mathsf{s_j} \mid \mathcal{C}_\mathsf{j}}^{\mathsf{j}} \qquad \overline{\Gamma \vdash \mathsf{s}_\ell \mid \mathcal{C}_\ell}^{\ell} \\ \mathcal{C} = \{\tau'' \prec \tau', \tau\ \textbf{introduces}\ \mathsf{cl\ meth}\ (\overline{\mathsf{cl_k\ var_k}}^{\mathsf{k}})\ \textbf{before}\ \mathsf{F}\} \cup \bigcup_\mathsf{j} \mathcal{C}_\mathsf{j} \cup \bigcup_\ell \mathcal{C}_\ell\end{array}}{\vdash_{\tau,\mathsf{F}} \textbf{refines method}\ \mathsf{cl\ meth}\ (\overline{\mathsf{cl_k\ var_k}}^{\mathsf{k}})\ \{\overline{\mathsf{s_j}}^{\mathsf{j}};\ \textbf{Super}();\ \overline{\mathsf{s}_\ell}^{\ell};\ \textbf{return}\ \mathsf{y};\} \mid \mathcal{C}} \text{ (WF-Refines-Method)}$$

$\boxed{\vdash_{\mathsf{F}} \mathsf{rcld} \mid \mathcal{C}}$ Class refinement well-formed in feature F subject to constraints $\mathcal{C}$

$$\frac{\begin{array}{c}\mathsf{dcl} \neq \mathsf{cl} \qquad \textbf{type}(\mathsf{dcl}) = \tau \qquad \overline{\vdash_\tau \mathsf{cl_k\ meth_k}\ (\overline{\mathsf{cl}_{\ell,\mathsf{k}}\ \mathsf{var}_{\ell,\mathsf{k}}}^{\ell})\ \mathsf{mb_k} \mid \mathcal{C}_\mathsf{k}}^{\mathsf{k}} \qquad \overline{\vdash_{\tau,\mathsf{F}} \mathsf{rmd_m} \mid \mathcal{C}'_\mathsf{m}}^{\mathsf{m}} \\ \xi = \bigcup_\mathsf{j}\{\mathsf{f_j} \notin \textbf{fields}(\textbf{parent}(\mathsf{dcl}))\} \qquad \upsilon = \bigcup_\mathsf{j}\{\mathsf{cl_j\ f_j}\ \textbf{unique in}\ \mathsf{dcl}\} \qquad \upsilon' = \bigcup_\mathsf{k}\{\mathsf{cl_k\ meth_k}\ (\overline{\mathsf{cl}_{\ell,\mathsf{k}}\ \mathsf{var}_{\ell,\mathsf{k}}}^{\ell})\ \textbf{unique in}\ \mathsf{dcl}\} \\ \xi' = \bigcup_\mathsf{k}\{\textbf{pmtype}(\mathsf{dcl}, \mathsf{meth_k}) = \overline{\mathsf{cl}_{\ell,\mathsf{k}}}^{\ell} \rightarrow \mathsf{cl_k}\}\end{array}}{\begin{array}{c}\vdash_\mathsf{F} \textbf{refines class}\ \mathsf{dcl}\ \textbf{extending}\ \mathsf{cl}\ \{\overline{\mathsf{cl_j\ f_j}}^{\mathsf{j}}; \overline{\mathsf{cl_k\ meth_k}\ (\overline{\mathsf{cl}_{\ell,\mathsf{k}}\ \mathsf{var}_{\ell,\mathsf{k}}}^{\ell,\mathsf{k}})\ \mathsf{mb_k}}^{\mathsf{k}};\ \overline{\mathsf{rmd}_{\ell,\mathsf{k}}}^{\ell,\mathsf{k}}\} \mid \bigcup_\mathsf{k} \mathcal{C}_\mathsf{k} \cup \bigcup_\mathsf{m} \mathcal{C}'_\mathsf{m} \cup \\ \{\textbf{defined}\ \mathsf{cl}, \overline{\textbf{defined}\ \mathsf{cl_j}}^{\mathsf{j}}, \mathsf{dcl}\ \textbf{introduced before}\ \mathsf{F},\} \cup \xi \cup \xi' \cup \upsilon \cup \upsilon'\end{array}}$$

(WF-Refines-Class)

Figure 2.8: Typing Rules for LFJ method and class refinements.

in PS or PS satisfies the constaints ξ. Compositional constraints on a feature F are satisfied when a feature with the appropriate introductions precedes F in PS. Uniqueness constraints are satisfied when no two features in PS introduce a member with the same name but different signatures to a class dcl. In LFJ, structural constraints are satisfied as in LJ, replacing uses of **path** with the CT function built by composition of the features in PS.

The compositional and uniqueness constraints guarantee that each step during the composition of a product specification builds an intermediate program. These programs need not be well-formed: they could rely on definitions which are introduced in a later feature or have classes used to satisfy typing constraints which could also be overwritten by a subsequent feature. For this reason, our typing rules only consider the final product specification, making no guarantees about the behavior of intermediate programs.

2.3.1 Soundness of the LFJ Type System

The soundness proof is based on successive refinements of the type systems of LJ and LFJ, ultimately reducing it to the proofs of progress and preservation of the original LJ type

$\boxed{\vdash \mathsf{P} \mid \mathcal{C}}$ Program well-formed subject to constraints

$$\frac{\overline{\vdash \mathsf{cld_k} \mid \mathcal{C}_\mathsf{k}}^{\mathsf{k}} \qquad \mathsf{P} = \overline{\mathsf{cld_k}}^{\mathsf{k}} \qquad \textbf{distinct names}\ (P)}{\vdash \mathsf{P} \mid \bigcup_\mathsf{k} \mathcal{C}_\mathsf{k}} \quad \text{(WF-Program)}$$

$\boxed{\vdash \mathsf{FD} \mid \mathcal{C}}$ Feature well-formed subject to constraints

$$\frac{\overline{\vdash \mathsf{cld_k} \mid \mathcal{C}_\mathsf{k}}^{\mathsf{k}} \qquad \overline{\vdash_\mathsf{F} \mathsf{rcld}_\ell \mid \mathcal{C}_\ell}^{\ell}}{\vdash \textbf{feature}\ \mathsf{F}\ \{\overline{\mathsf{cld_k}}^{\mathsf{k}}\overline{\mathsf{rcld}_\ell}^{\ell}\} \mid \mathbf{In}_\mathsf{F} \Rightarrow \bigcup_\mathsf{k} \mathcal{C}_\mathsf{k} \cup \bigcup_\ell \mathcal{C}_\ell} \quad \text{(WF-Feature)}$$

$\boxed{\vdash \mathsf{FT} \mid \mathcal{C}}$ Feature Table well-formed subject to constraints

$$\frac{\mathsf{FT} = \{\overline{\mathsf{FD_k}}^{\mathsf{k}}\} \qquad \overline{\vdash \mathsf{FD_k} \mid \mathcal{C}_\mathsf{k}}^{\mathsf{k}}}{\vdash \mathsf{FT} \mid \bigcup_k \mathcal{C}_k} \quad \text{(WF-Feature-Table)}$$

Figure 2.9: Typing Rules for LFJ Programs and Features.

system [SSP07]. We first show that the constraint-based LJ type system is equivalent to the original LJ type system, in that a program with unique class names and an acyclic class hierarchy satisfies its constraints if and only if it is well-formed according to the original typing rules. We then show that any LFJ product specification will build a well-formed LJ program if it satisfies the feature table constraints generated by the constraint-based LFJ type system. We have formalized in the Coq proof assistant the syntax and semantics of LJ and LFJ presented in the previous section, as well as all of the soundness proofs that follow. For this reason, the following sections elide many of the bookkeeping details, instead presenting sketches of the major pieces of the soundness proofs.

Theorem 2.3.1 (Soundness of constraint-based LJ Type System)**.** *Let* P *be an LJ program with distinct class names and an acyclic, well-founded class hierarchy. Let* $\mathcal{C}$ *be the set of constraints generated by a class* cld *in* P*.* cld *is well-formed if and only if* P *satisfies* $\mathcal{C}$*:* $\mathsf{P} \vdash \mathsf{cld} \leftrightarrow \mathsf{P} \models \mathcal{C}$ *where* $\vdash \mathsf{cld} \mid \mathcal{C}$*.*

Proof. The two key pieces of this proof are: showing that satisfaction of each of the constraints guarantees that the corresponding judgement holds, and that there is a one-to-one correspondence between the constraints generated by the typing rules in Figure 2.8 and the external premises used in the declarative LJ type system. The former is straightforward ex-

$$\frac{\mathbf{ftype}(\mathsf{P},\tau_1,\mathsf{f}) = \tau_3 \qquad \tau_2 \in \mathbf{path}(\mathsf{P},\tau_3)}{\mathsf{P} \models \tau_2 \prec \mathbf{ftype}(\tau_1,\mathsf{f})}$$

$$\frac{\mathbf{ftype}(\mathsf{P},\tau_1,\mathsf{f}) = \tau_3 \qquad \tau_3 \in \mathbf{path}(\mathsf{P},\tau_2)}{\mathsf{P} \models \mathbf{ftype}(\tau_1,\mathsf{f}) \prec \tau_2}$$

$$\frac{\mathbf{mtype}(\mathsf{P},\tau,\mathsf{m}) = \overline{\pi'_k}^{k} \to \pi' \qquad \pi' \in \mathbf{path}(\mathsf{P},\pi) \qquad \overline{\pi_k \in \mathbf{path}(\mathsf{P},\pi'_k)}^{k}}{\mathsf{P} \models \mathbf{mtype}(\tau,\mathsf{m}) \prec \overline{\pi_k}^{k} \to \pi}$$

$$\frac{\mathbf{type}(\mathsf{cl}) \in \mathbf{path}(\mathsf{P},\mathbf{type}(\mathsf{cl}))}{\mathsf{P} \models \mathbf{defined}(\mathsf{cl})}$$

$$\frac{\tau_2 \in \mathbf{path}(\mathsf{P},\tau_1)}{\mathsf{P} \models \tau_1 \prec \tau_2}$$

$$\frac{\mathbf{ftype}(\mathsf{P},\mathbf{parent}(\mathsf{dcl}),\mathsf{f}) = \bot}{\mathsf{P} \models \mathsf{f} \notin \mathbf{fields}(\mathbf{parent}(\mathsf{dcl}))}$$

$$\frac{\mathbf{mtype}(\mathsf{P},\mathbf{parent}(\mathsf{dcl}),\mathsf{m}) = \bot \ \vee \ \mathbf{mtype}(\mathsf{P},\mathbf{parent}(\mathsf{dcl}),\mathsf{m}) = \tau}{\mathsf{P} \models \mathbf{pmtype}(\mathsf{dcl},\mathsf{m}) = \tau}$$

$$\frac{\mathsf{FP} = \overline{\mathsf{A}_k}^{k}\mathsf{F}\overline{\mathsf{B}_\ell}^{\ell}\mathsf{H}\overline{\mathsf{C}_j}^{j} \qquad \tau.\mathsf{ms} \in \mathsf{H} \qquad \tau \notin \mathbf{introductions}(\overline{\mathsf{B}_\ell}^{\ell})}{\mathsf{PS} \models \tau \ \mathbf{introduces}\ \mathsf{ms}\ \mathbf{before}\ \mathsf{F}}$$

$$\frac{\mathsf{PS} = \overline{\mathsf{A}_k}^{k}\mathsf{F}\overline{\mathsf{B}_\ell}^{\ell}\mathsf{H}\overline{\mathsf{C}_j}^{j} \qquad \mathbf{dcl} \in \mathsf{H}}{\mathsf{PS} \models \mathsf{dcl}\ \mathbf{introduced\ before}\ \mathsf{F}}$$

$$\frac{\mathbf{type}(\mathsf{dcl}) = \tau \qquad \forall \mathsf{A},\mathsf{B} \in \mathsf{PS}, \tau.\mathsf{cl}_1\ \mathsf{f} \in \mathsf{A} \wedge \tau.\mathsf{cl}_2\ \mathsf{f} \in \mathsf{B} \to \mathsf{cl}_1 = \mathsf{cl}_2}{\mathsf{PS} \models \mathsf{cl}\ \mathsf{f}\ \mathbf{unique\ in}\ \mathsf{dcl}}$$

$$\frac{\mathbf{type}(\mathsf{dcl}) = \tau \qquad \mathsf{ms}_1 = \mathsf{cl}\ \mathsf{m}\ (\overline{\mathsf{vd}_k}^{k}) \qquad \mathsf{ms}_2 = \mathsf{cl}'\ \mathsf{m}\ (\overline{\mathsf{vd}'_k}^{k}) \qquad \forall \mathsf{A},\mathsf{B} \in \mathsf{PS}, \tau.\mathsf{ms}_1 \in \mathsf{A} \wedge \tau.\mathsf{ms}_2 \in \mathsf{B} \to \mathsf{ms}_1 = \mathsf{ms}_2}{\mathsf{PS} \models \mathsf{cl}\ \mathsf{m}\ (\overline{\mathsf{vd}_k}^{k})\ \mathbf{unique\ in}\ \mathsf{dcl}}$$

$$\frac{\mathsf{F} \notin \mathsf{PS}}{\mathsf{PS} \models \mathbf{In}_\mathsf{F} \Rightarrow \overline{\xi_k}^{k}}$$

$$\frac{\mathsf{F} \in \mathsf{PS} \qquad \overline{\mathsf{PS} \models \xi_k}^{k}}{\mathsf{PS} \models \mathbf{In}_\mathsf{F} \Rightarrow \overline{\xi_k}^{k}}$$

Figure 2.10: Satisfaction of typing constraints.

cept for the subtyping constraint, which relies on the **path** function to check for satisfaction. We can prove their equivalence by induction on the derivation of the subtyping judgement in one direction and induction on the length of the path in the other. We can then show that the two type systems are equivalent by examination of the structure of P. At each level of the typing rules, the structural premises are identical and each of the external premises of the rules is represented in the set of constraints. As a result of the previous argument, satisfaction of the constraints guarantees that premises of the typing rules hold for each structure in P. Having shown the two type systems are equivalent, the proofs of progress and preservation for the constraint-based type system follow immediately. □

Theorem 2.3.2 (Soundness of LFJ Type System)**.** *Let* PS *be an LFJ product specification for feature table* FT*and* $\mathcal{C}$ *be a set of constraints such that* $\vdash \mathsf{FT} \mid \mathcal{C}$*. If PS* $\models \mathcal{C}$ *and* ***Object*** *is in the path of every class introduced by a feature in* PS*, then the composition of the features in* PS *produces a valid, well-formed LJ program.*

Proof. This proof is decomposed into three key lemmas, corresponding to the three kinds of typing constraints:

(i) Composition of the features in PS produces a valid LJ program, P.

For each class or method refinement of a feature F in PS, a composition constraint is generated by the LFJ typing rules. Each of these are satisfied according to the definition in Figure 2.10, allowing us to conclude that a feature with appropriate declarations appears before F in PS. Each of these declarations will appear in the program generated by the features preceding F, allowing us to conclude that the composition succeeds for each feature in PS.

(ii) P is typeable in the constraint-based LJ type system with constraints $\mathcal{C}'$.

In essence, we must show that the premises of the constraint-based LJ typing judgements hold. Our assumption that each class in PS is a descendant of **Object** ensures that P has an acyclic, well-founded class hierarchy. The premises for the LJ methods and statements are identical, leaving class typing rules for us to consider. The LJ typing rules require that the method and field names for a class be distinct, but these premises are removed by the LFJ typing rules, as the members of a class are not finalized until after composition. This requirement is instead enforced by the uniqueness constraints in Figure 2.10, which are satisfied only when a method or field name is introduced by a single feature. Since $\mathsf{PS} \models \mathcal{C}$, it follows that the premises of the LJ typing rules hold for P and that there exists some set of constraints $\mathcal{C}'$ such that $\vdash \mathsf{P} \mid \mathcal{C}'$.

(iii) P satisfies the constraints in $\mathcal{C}'$ and is thus a well-formed LJ program.

We break this proof into two sublemmas:

(a) $\mathcal{C}' \subseteq \mathcal{C}$.

The key observation for this proof is that every class, method, and statement in P originated from some feature in PS. Since $\mathsf{PS} \models \mathcal{C}$, it follows that PS satisfies the structural constraints of each of its features. The most interesting case is for the constraints generated by method bodies: a statement contained in a method body can come from either the initial introduction of that method or advice added by a method refinement. In either case, the statement was included in some feature in PS and thus generated some set of constraints in $\mathcal{C}$. Because method signatures are fixed across refinement, the context used in typing both kinds of statements is the same as that used for the method in the final composition.

(b) For any structural constraint $\mathcal{K}$, if $\mathsf{PS} \models \mathcal{K}$, then $\mathsf{P} \models \mathcal{K}$.

This reduces to showing that class declaration returned by $\mathsf{CT(dcl)}$ is the same as the class with that identifier in P. This follows from tracing the definition of the CT function down to the final introduction of dcl in the product line. From here, we know that this class appears in the program synthesized from the product specification starting with this feature. Further refinements of this class are reflected in the + operator used recursively to build $\mathsf{CT(dcl)}$; each refinement succeeds by (i) above. Since the two functions are the same, the helper functions which call **path** in P (i.e. **ftype**, **mtype**) and those that use CT in PS return the same values. We can thus conclude that the satisfaction judgements for PS and P are equivalent.

All constraints in $\mathcal{C}'$ appear in $\mathcal{C}$, so $\mathsf{PS} \models \mathcal{C}'$. By (b) above, it follows that $\mathsf{P} \models \mathcal{C}'$. P must therefore be a well-formed LJ program by Theorem 2.3.1. $\square$

2.3.2 Type System Origins

The genesis of this type system was a system developed by Thaker et al. [TBKC07] which generated the implementation constraints of an AHEAD product line of Java programs by examining field, method, and class references in feature definitions. Analysis of existing product lines using this system detected previously unknown errors in their feature models. The authors identified five properties that are necessary for a composition to be well-typed, and gave constraints which a product specification must satisfy for the properties to hold. The constraints used by the LFJ type system are the "properties" in our approach and the translation from our type system's constraints to propositional formulas builds the product specification "constraints" used by Thaker et al. Because we use the type system to generate these constraints, we are able to leverage the proofs of soundness to guarantee safe composition by using constraints that are necessary and sufficient for type-safety.

Chapter 3

Feature Modularity in Programming Language Metatheory

The type system of LFJ from the previous chapter is used to build semantic feature modules whose proofs are typing derivations for the associated syntactic modules. Because this analysis is decidable, type safety of a program specified by a feature selection can be easily verified without considering the type safety of the included feature modules. This limits the need for reusable semantic modules[1], although the semantic interfaces still enhance reusability of syntactic modules by providing a clear compatibility specification.

If semantic modules contain proofs in a non-decideable logic, reusability becomes more important. The alternative is to have users craft proofs for each feature selection—leading to a potentially pain-staking replication of proofs. Proper semantic interfaces specify which features are compatible, clearly indicating where proofs must be patched or extended. In much the same way that modularity allows software to be engineered for reuse in related products, semantic modularity allows proofs to be reused, maintained, and extended[2].

[1]Chapter 6 demonstrates this is key for efficiently typechecking an entire product line.

[2]The observation that theorem proving can enjoy the same benefits from engineering as software development is another application of the Curry-Howard isomorphism between proofs and programs [Cur34, How80].

The programming language literature is replete with examples of both syntactic and semantic reuse. Taking a core language such as *Featherweight Java*(FJ) [IPW01] and extending it with a single feature is standard practice – the original FJ paper itself introduced *Featherweight Generic Java* (FGJ), a modified version of FJ with support for generics. These pen-and-paper formalizations employ a cut-paste-patch approach to syntactic reuse by modifying existing definitions and inserting new ones. Semantic reuse is trickier, as patching up existing proofs for these extended definitions requires care, making integration of new features something of an art. In part because the natural decomposition of a language into features cuts across the modularity boundaries of theorem provers, this cut-paste-patch approach to language extension has carried over to mechanizations of programming language metatheory. As an example, Leroy's three person-year CompCert verified compiler project [Ler09] consists of eight intermediate languages in addition to the source and target languages, many of which are minor variations of each other.

This approach is perhaps the best we can hope for in pen-and-paper formalizations, which lack the means for enforcement of modularity boundaries. In contrast, proof assistants enable the construction of true modules with statically-enforceable interfaces. For this reason, proper syntactic and semantic modularization techniques hold the potential to transform mechanization of semantics from an important confidence-booster into a powerful vehicle for reuse. The authors of the PoplMark challenge [A^{+}05] noted the importance of *reuse of components* in facilitating experimentation and enabling researchers to share language designs when they identified it as a key challenge in mechanizing metatheory.

Using the modularization of the pen-and-paper formalization of FGJ as an example, this chapter identifies common forms of variability in language design in order to identify the sorts of variation points that semantic metatheory modules must support. Having identified the fundamental concepts needed to support extensible language formalizations, subsequent chapters tackle the engineering challenges of supporting these variation points in the Coq proof assistant.

3.1 A Motivating Example

Adding generics to the calculus of FJ produces the FGJ calculus, weaving a number of changes throughout the syntax and semantics of FJ. The left-hand column of Figure 3.1 presents a subset of the syntax of FJ, the rules which formalize the subtyping relation which establishes the inheritance hierarchy, and the typing rule that ensures expressions for object creation are well-formed. The corresponding definitions for FGJ are in the right-hand column.

FJ Expression Syntax		**FGJ Expression Syntax**
$\mathtt{e} ::= \mathtt{x}$ $\mid \mathtt{e.f}$ $\mid \mathtt{e.m}\ (\bar{\mathtt{e}})$ $\mid \mathtt{new\ C}(\bar{\mathtt{e}})$ $\mid (\mathtt{C})\ \mathtt{e}$	$\Longmapsto$	$\mathtt{e} ::= \mathtt{x}$ $\mid \mathtt{e.f}$ $\mid \mathtt{e.m}\langle\bar{\mathtt{T}}\rangle^{\beta}\ (\bar{\mathtt{e}})$ $\mid \mathtt{new\ C}\ \langle\bar{\mathtt{T}}\rangle^{\beta}\ (\bar{\mathtt{e}})$ $\mid (\mathtt{C}\ \langle\bar{\mathtt{T}}\rangle^{\beta})\ \mathtt{e}$
FJ Subtyping $\qquad \mathtt{T} <: \mathtt{T}$		**FGJ Subtyping** $\qquad \Delta^{\delta} \vdash \mathtt{T} <: \mathtt{T}$
$\dfrac{\mathtt{S}<:\mathtt{T} \qquad \mathtt{T}<:\mathtt{V}}{\mathtt{S}<:\mathtt{V}}$ (S-Trans) $\mathtt{T}<:\mathtt{T}$ (S-Refl) $\dfrac{\mathtt{class\ C\ extends\ D}\ \{\ldots\}}{\mathtt{C}<:\mathtt{D}}$ (S-Dir)	$\Longmapsto$	$\Delta \vdash \mathtt{X}<:\Delta(\mathtt{X})$ (GS-Var)$^{\alpha}$ $\dfrac{\Delta^{\delta} \vdash \mathtt{S}<:\mathtt{T} \qquad \Delta^{\delta} \vdash \mathtt{T}<:\mathtt{V}}{\Delta^{\delta} \vdash \mathtt{S}<:\mathtt{V}}$ (GS-Trans) $\Delta^{\delta} \vdash \mathtt{T}<:\mathtt{T}$ (GS-Refl) $\dfrac{\mathtt{class\ C}\ \langle\bar{\mathtt{X}} \triangleleft \bar{\mathtt{N}}\rangle^{\beta}\ \mathtt{extends\ D}\ \langle\bar{\mathtt{V}}\rangle^{\beta}\ \{\ldots\}}{\Delta^{\delta} \vdash \mathtt{C}\ \langle\bar{\mathtt{T}}\rangle^{\beta} <: [\bar{\mathtt{T}}/\bar{\mathtt{X}}]^{\eta}\mathtt{D}\ \langle\bar{\mathtt{V}}\rangle^{\beta}}$ (GS-Dir)
FJ New Typing $\qquad \Gamma \vdash \mathtt{e} : \mathtt{T}$		**FGJ New Typing** $\qquad \Delta;^{\delta}\Gamma \vdash \mathtt{e} : \mathtt{T}$
$\dfrac{\mathit{fields}(\mathtt{C}) = \bar{\mathtt{V}}\ \bar{\mathtt{f}} \qquad \Gamma \vdash \bar{\mathtt{e}} : \bar{\mathtt{U}} \qquad \bar{\mathtt{U}}<:\bar{\mathtt{V}}}{\Gamma \vdash \mathtt{new\ C}(\bar{\mathtt{e}}) : \mathtt{C}}$ (T-New)	$\Longmapsto$	$\dfrac{\Delta \vdash \mathtt{C}\langle\bar{\mathtt{T}}\rangle^{\gamma} \qquad \mathit{fields}(\mathtt{C}\ \langle\bar{\mathtt{T}}\rangle^{\beta}) = \bar{\mathtt{V}}\ \bar{\mathtt{f}} \qquad \Delta;^{\delta}\Gamma \vdash \bar{\mathtt{e}} : \bar{\mathtt{U}} \qquad \Delta^{\delta} \vdash \bar{\mathtt{U}}<:\bar{\mathtt{V}}}{\Delta;^{\delta}\Gamma \vdash \mathtt{new\ C}\ \langle\bar{\mathtt{T}}\rangle^{\beta}(\bar{\mathtt{e}}) : \mathtt{C}}$ (GT-New)

Figure 3.1: Selected FJ Definitions with FGJ Changes Highlighted

The categories of changes are shaded and tagged in Figure 3.1 with Greek letters:

α. *Adding new rules or pieces of syntax.* FGJ adds type variables to parameterize classes and methods. The subtyping relation adds the GS-Var rule for this new kind of type.

β. *Modifying existing syntax.* FGJ adds type parameters to method calls, object creation, casts, and class definitions.

γ. *Adding new premises to existing typing rules.* The updated GT-New rule includes a new premise requiring that the type of a new object must be well-formed.

δ. *Extending judgment signatures.* The added rule GS-Var looks up the bound of a type variable using a typing context, Δ. This context must be added to the signature of the subtyping relation, transforming all occurrences to a new ternary relation.

η. *Modifying premises and conclusions in existing rules.* The type parameters used for the parent class D in a class definition are instantiated with the parameters used for the child in the conclusion of GS-Dir.

In addition to these definitions, FJ and FGJ include proofs of progress and preservation for their type systems. With each change to a definition, these proofs must also be updated. As with the changes to definitions in Figure 3.1, these changes are threaded throughout existing proofs. Consider the related proofs in Figure 3.2 of a lemma used in the proof of progress for both languages. These lemmas are used in the same place in the proof of progress and are structurally similar, proceeding by induction on the derivation of the subtyping judgment. The proof for FGJ has been adapted to reflect the changes that were made to its definitions. These changes are highlighted in Figure 3.2 and marked with the kind of definitional change that triggered the update. Throughout the lemma, the signature of the subtyping judgment has been altered include a context for type variables$^{\delta}$. The statement of the lemma now uses the auxiliary bound function, due to a modification to the premises of the typing rule for field lookup$^{\eta}$. These changes are not simply syntactic: both affect the applications of the inductive hypothesis in the GS-Trans case. The proof must now include a case for the added GS-Var subtyping rule$^{\alpha}$. The case for GS-Dir requires the most drastic change, as the existing proof for that case is modified to include an additional statement about the behavior of bound.

As more features are added to a language, its metatheoretic proofs of correctness grow in size and complexity. Each different feature selection produces a new language with its own syntax, type system, and operational semantics. While the proof of type safety is similar for each language, (potentially subtle) changes occur throughout the proof depending on the features included. Modularizing the type safety proof into distinct features allows the type safety proof for each language variant to be built from a common set of proofs. There is no need to manually maintain separate proofs for each language variant. Importantly, this allows language designers to add new features to an existing language in a structured way, exploiting existing proofs to build more feature-rich languages that are semantically correct.

The following sections detail the structural changes to proofs that each kind of extension to a language's syntax and semantics outlined above requires. Intuitively, a semantic module's exported proofs abstract over semantic properties of the imported VPs of the corresponding syntactic module, forming a semantic imports interface. Replacing these assumptions with proofs for a concrete definition enables reuse of potentially complex proofs for any valid definition. Proper modules guarantee that the composite proof is guaranteed to hold for any composed language, as long as an extension provides the necessary proofs to satisfy these assumptions. Proofs for a composed language are thus built through module composition, with a straightforward interface check sufficing to certify their correctness.

FJ Fields of a Supertype		**FGJ Fields of a Supertype**
Lemma 3.1.1. *If* $\mathtt{S} <: \mathtt{T}$ *and* $\mathtt{fields(T)} = \bar{\mathtt{T}}\ \bar{\mathtt{f}}$, *then* $\mathtt{fields(S)} = \bar{\mathtt{S}}\ \bar{\mathtt{g}}$ *and* $\mathtt{S}_i = \mathtt{T}_i$ *and* $\mathtt{g}_i = \mathtt{f}_i$ *for all* $i \leqslant \#(\mathtt{f})$.	$\Mapsto$	**Lemma 3.1.2.** *If* $\Delta^{\delta} \vdash \mathtt{S} <: \mathtt{T}$ *and* $\mathtt{fields(}\mathtt{bound}_{\Delta}\mathtt{(T)}^{\eta}\mathtt{)} = \bar{\mathtt{T}}\ \bar{\mathtt{f}}$, *then* $\mathtt{fields(}\mathtt{bound}_{\Delta}\mathtt{(S)}^{\eta}\mathtt{)} = \bar{\mathtt{S}}\ \bar{\mathtt{g}}$, $\mathtt{S}_i = \mathtt{T}_i$ *and* $\mathtt{g}_i = \mathtt{f}_i$ *for all* $i \leqslant \#(\mathtt{f})$.
Proof. By induction on the derivation of $\mathtt{S} <: \mathtt{T}$		*Proof.* By induction on the derivation of $\Delta^{\delta} \vdash \mathtt{S} <: \mathtt{T}$
		Case GS-Var$^{\alpha}$ $\mathtt{S} = \mathtt{X}$ and $\mathtt{T} = \Delta(\mathtt{X})$. Follows immediately from the fact that $\mathtt{bound}_{\Delta}(\Delta(\mathtt{X})) = \Delta(\mathtt{X})$ by the definition of $\mathtt{bound}$.
Case S-Refl $\mathtt{S} = \mathtt{T}$. Follows immediately.	$\Mapsto$	**Case GS-Refl** $\mathtt{S} = \mathtt{T}$. Follows immediately.
Case S-Trans $\mathtt{S} <: \mathtt{V}$ and $\mathtt{V} <: \mathtt{T}$. By the inductive hypothesis, $\mathtt{fields(V)} = \bar{\mathtt{V}}\ \bar{\mathtt{h}}$ and $\mathtt{V}_i = \mathtt{T}_i$ and $\mathtt{h}_i = \mathtt{f}_i$ for all $i \leqslant \#(\mathtt{f})$. Again applying the inductive hypothesis, $\mathtt{fields(S)} = \bar{\mathtt{S}}\ \bar{\mathtt{g}}$ and $\mathtt{S}_i = \mathtt{V}_i$ and $\mathtt{g}_i = \mathtt{h}_i$ for all $i \leqslant \#(\mathtt{h})$. Since $\#(\mathtt{f}) \leqslant \#(\mathtt{h})$, the conclusion is immediate.	$\Mapsto$	**Case GS-Trans** $\Delta^{\delta} \vdash \mathtt{S} <: \mathtt{V}$ and $\Delta^{\delta} \vdash \mathtt{V} <: \mathtt{T}$. By the inductive hypothesis, $\mathtt{fields(}\mathtt{bound}_{\Delta}\mathtt{(V)}^{\eta}\mathtt{)} = \bar{\mathtt{V}}\ \bar{\mathtt{h}}$ and $\mathtt{V}_i = \mathtt{T}_i$ and $\mathtt{h}_i = \mathtt{f}_i$ for all $i \leqslant \#(\mathtt{f})$. Again applying the inductive hypothesis, $\mathtt{fields(}\mathtt{bound}_{\Delta}\mathtt{(S)}^{\eta}\mathtt{)} = \bar{\mathtt{S}}\ \bar{\mathtt{g}}$ and $\mathtt{S}_i = \mathtt{V}_i$ and $\mathtt{g}_i = \mathtt{h}_i$ for all $i \leqslant \#(\mathtt{h})$. Since $\#(\mathtt{f}) \leqslant \#(\mathtt{h})$, the conclusion is immediate.
Case S-Dir $\mathtt{S} = \mathtt{C}$, $\mathtt{T} = \mathtt{D}$ $\mathtt{class\ C\ extends\ D}\ \{\bar{\mathtt{S}}\ \bar{\mathtt{g}}; \ldots\}$. By the rule F-Class, $\mathtt{fields(C)} = \bar{\mathtt{U}}\ \bar{\mathtt{f}}; \bar{\mathtt{S}}\ \bar{\mathtt{g}}$, where $\bar{\mathtt{U}}\ \bar{\mathtt{f}} = \mathtt{fields(D)}$, from which the conclusion is immediate.	$\Mapsto$	**Case GS-Dir** $\mathtt{S} = \mathtt{C}\langle\bar{\mathtt{T}}\rangle^{\beta}$, $\mathtt{T} = [\bar{\mathtt{T}}/\bar{\mathtt{X}}]^{\eta}\mathtt{D}\langle\bar{\mathtt{V}}\rangle^{\beta}$, $\mathtt{class\ C}\langle\bar{\mathtt{X}} \triangleleft \bar{\mathtt{N}}\rangle^{\beta}\ \mathtt{extends\ D}\langle\bar{\mathtt{V}}\rangle^{\beta}\ \{\bar{\mathtt{S}}\ \bar{\mathtt{g}}; \ldots\}$. By the rule F-Class, $\mathtt{fields(C}\langle\bar{\bar{\mathtt{T}}}\rangle^{\beta}\mathtt{)} = \bar{\mathtt{U}}\ \bar{\mathtt{f}};\ [\bar{\mathtt{T}}/\bar{\mathtt{X}}]^{\eta}\bar{\mathtt{S}}\ \bar{\mathtt{g}}$, where $\bar{\mathtt{U}}\ \bar{\mathtt{f}} = \mathtt{fields(}[\bar{\mathtt{T}}/\bar{\mathtt{X}}]^{\eta}\ \mathtt{D}\ \langle\bar{\bar{\mathtt{V}}}\rangle^{\beta}\mathtt{)}$. By definition, $\mathtt{bound}_{\Delta}\mathtt{(V)} = \mathtt{V}$ for all non-variable types $\mathtt{V}^{\eta}$, from which the conclusion is immediate.

Figure 3.2: An Example FJ Proof with FGJ Changes Highlighted

3.2 The Features of FGJ

The remainder of this chapter uses *core Featherweight Java* (cFJ) language as the *base* feature of the core language to which extensions are added. cFJ is a cast-free variant of FJ. (This omission is not without precedent, as other core calculi for Java [SSP07] omit casts). There are also three *optional* features which extend the base or other features:

cFJ	core Featherweight Java
Cast	adds casts to expressions
Interface	adds interfaces
Generic	adds parameterized types

Cast + cFJ specifies the original FJ, while other feature selections specify other FJ variants:

cFJ	// Core FJ
Cast + cFJ	// Original FJ [IPW01]
Interface + cFJ	// Core FJ with Interfaces
Interface + Cast + cFJ	// Original FJ with Interfaces
Generic + cFJ	// Core Featherweight Generic Java
Generic + Cast + cFJ	// Original FGJ [IPW01]
Generic + Interface + cFJ	// core Generic FJ with Generic Interfaces
Generic + Interface + Cast + cFJ	// FGJ with Generic Interfaces

The complete set of non-empty feature and feature interactions for our example is:

Module	**Description**
cFJ	core Featherweight Java
Cast	cast
Interface	interfaces
Generic	generics
Generic#Interface	generic and interface interactions
Generic#Cast	generic and cast interactions

3.3 Decomposing a Language into Features

The syntax, operational semantics, type system and metatheoretic proofs of a pen-and-paper formalization of a programming language are written in distinct languages. Syntax uses BNF, operational semantics and type systems use declarative rules, and metatheoretic proofs use in english (or french, russian, spanish, etc.). Despite these different representations, two changes categorize the variability required by language features: adding new definitions and modifying existing definitions. The following sections illustrate how variation points allowing these changes can be used to address the design challenges in modularizing features.

3.3.1 Language Syntax

Language syntax is usually expressed as a context-free grammar using Backus–Naur Form. Figure 3.3a shows the BNF for cFJ expressions, Figure 3.3b the production that the Cast feature adds to cFJ's BNF, and Figure 3.3c the syntax of $\mathsf{FJ} = \mathsf{Cast} + \mathsf{cFJ}$ (Figure 3.1). The syntax resulting from the composition of Cast and cFJ is the union of the productions of each. Intuitively, the nonterminal e serves as the variation point for these rules such that composition combines all exported definitions of e.

$$\boxed{\begin{array}{lcl} \mathsf{e} & ::= & \mathsf{x} \\ & | & \mathsf{e.f} \\ & | & \mathsf{e.m}(\bar{\mathsf{e}}) \\ & | & \mathsf{new\ C}(\bar{\mathsf{e}})\ ; \end{array}}$$

(a)

$$\boxed{\begin{array}{lcl} \mathsf{e} & ::= & (\mathsf{C})\ \mathsf{e}\ ; \end{array}}$$

(b)

$$\boxed{\begin{array}{lcl} \mathsf{e} & ::= & \mathsf{x} \\ & | & \mathsf{e.f} \\ & | & \mathsf{e.m}(\bar{\mathsf{e}}) \\ & | & \mathsf{new\ C}(\bar{\mathsf{e}}) \\ & | & (\mathsf{C})\ \mathsf{e}\ ; \end{array}}$$

(c)

Figure 3.3: Union of Grammars

To see how a feature can modify existing syntax, consider the impact of adding the

Generics feature to cFJ and Cast: type parameters must be added to the expression syntax of both method calls and class types. Figure 3.4a-b shows the syntax for method calls from the cFJ expression grammar and the syntax for class types from the cFJ and Cast expression grammars with the $\mathsf{TP_m}$ and $\mathsf{TP_t}$ VPs added to each respectively. These productions are empty by default, as indicated in Figure 3.4a. Figure 3.4c shows the union of the revised Cast and cFJ expression grammars. Since $\mathsf{TP_m}$ and $\mathsf{TP_t}$ are empty, these productions can be inlined to produce the grammar in Figure 3.3c.

$$
\begin{array}{rcl}
\mathsf{e} & ::= & \mathsf{x} \\
 & | & \mathsf{e.f} \\
 & | & \mathsf{TP_m}\ \mathsf{e.m}\ (\bar{\mathsf{e}}) \\
 & | & \mathsf{new}\ (\ \mathsf{TP_t}\ \mathsf{C})\ (\bar{\mathsf{e}}); \\
\mathsf{TP_m} & ::= & \epsilon; \\
\mathsf{TP_t} & ::= & \epsilon;
\end{array}
$$

(a)

$$
\mathsf{e}\ ::=\ (\ \mathsf{TP_t}\ \mathsf{C})\ \mathsf{e};
$$

(b)

$$
\begin{array}{rcl}
\mathsf{e} & ::= & \mathsf{x} \\
 & | & \mathsf{e.f} \\
 & | & \mathsf{TP_m}\ \mathsf{e.m}\ (\bar{\mathsf{e}}) \\
 & | & \mathsf{new}\ (\ \mathsf{TP_t}\ \mathsf{C})\ (\bar{\mathsf{e}}) \\
 & | & (\ \mathsf{TP_t}\ \mathsf{C})\ \mathsf{e}; \\
\mathsf{TP_m} & : & \epsilon; \\
\mathsf{TP_t} & : & \epsilon;
\end{array}
$$

(c)

Figure 3.4: Modification of Grammars

Alternatively, defining $\mathsf{TP_m}$ and $\mathsf{TP_t}$ to be lists of type parameters builds the syntax of FGJ, as shown in Figure 3.5a. Intuitively, the Generic feature exports these definitions, so that the syntax for expressions specified by Generic + Cast + cFJ is shown in Figure 3.5b.

$$
\begin{array}{rcl}
\mathsf{TP_m} & ::= & \langle\bar{\mathsf{T}}\rangle; \\
\mathsf{TP_t} & ::= & \langle\bar{\mathsf{T}}\rangle;
\end{array}
$$

(a)

$$
\begin{array}{rcl}
\mathsf{e} & ::= & \mathsf{x} \\
 & | & \mathsf{e.f} \\
 & | & \langle\bar{\mathsf{T}}\rangle\ \mathsf{e.m}\ (\bar{\mathsf{e}}) \\
 & | & \mathsf{new}\ (\langle\bar{\mathsf{T}}\rangle\ \mathsf{C})\ (\bar{\mathsf{e}}) \\
 & | & (\langle\bar{\mathsf{T}}\rangle\ \mathsf{C})\ \mathsf{e};
\end{array}
$$

(b)

Figure 3.5: The Effect of Adding Generics to Expressions

Replacing an empty production with a non-empty one is a standard programming practice in frameworks (e.g. EJB [MH01]). Framework hook methods are initially empty and users can override them with a definition that is specific to their context. The same engineering approach applies here as well.

3.3.2 Reduction and Typing Rules

Declarative rules define the judgments that form the operational semantics and type system of a language. Figure 3.6a shows the typing rules for cFJ expressions, Figure 3.6b the rule that the Cast feature adds, and Figure 3.6c the composition (union) of these rules, defining the typing rules for FJ. The typing judgement used in the subderivations is the variation point in Figure 3.6a-b in much the same way as the nonterminal e was the variation point for the productions in Figure 3.3a.

$$\boxed{\begin{array}{c} \dfrac{\begin{array}{c}\mathsf{fields}(\mathsf{C}) = \overline{\mathsf{V}}\ \bar{\mathsf{f}} \\ \Gamma \vdash \bar{\mathsf{e}} : \overline{\mathsf{U}} \qquad \overline{\mathsf{U}} <: \overline{\mathsf{V}}\end{array}}{\Gamma \vdash \mathsf{new}\ \mathsf{C}(\bar{\mathsf{e}}) : \mathsf{C}}\ (\text{T-New}) \\ \vdots \end{array}}$$

(a)

$$\boxed{\begin{array}{c} \dfrac{\Gamma \vdash \mathsf{e}_0 : \mathsf{D} \qquad \mathsf{D} <: \mathsf{C}}{\Gamma \vdash \mathsf{e}_0 : \mathsf{C}} \\ \hfill (\text{T-UCast}) \end{array}}$$

(b)

$$\boxed{\begin{array}{c} \dfrac{\begin{array}{c}\mathsf{fields}(\mathsf{C}) = \overline{\mathsf{V}}\ \bar{\mathsf{f}} \\ \Gamma \vdash \bar{\mathsf{e}} : \overline{\mathsf{U}} \qquad \overline{\mathsf{U}} <: \overline{\mathsf{V}}\end{array}}{\Gamma \vdash \mathsf{new}\ \mathsf{C}(\bar{\mathsf{e}}) : \mathsf{C}}\ (\text{T-New}) \\ \vdots \\ \dfrac{\Gamma \vdash \mathsf{e}_0 : \mathsf{D} \qquad \mathsf{D} <: \mathsf{C}}{\Gamma \vdash \mathsf{e}_0 : \mathsf{C}} \\ \hfill (\text{T-UCast}) \end{array}}$$

(c)

Figure 3.6: Union of Typing Rules

Modifying existing rules is analogous to language syntax. There are three kinds of VPs for rules: (a) predicates that extend the premise of a rule, (b) relational holes which extend a judgement's signature, and (c) functions that transform existing premises and conclusions. Predicate and relational holes are empty by default, while the identity function is the default for functions. This applies to both the reduction rules that define a language's operational semantics and the typing rules that define its type system.

To build the typing rules for FGJ, the Generic feature adds non-empty definitions for the $\mathsf{WF_c}(\mathsf{D}, \mathsf{TP_t}\ \mathsf{C})$ predicate and for the D VP in the cFJ typing definitions. (Compare Figure 3.6a to its VP-extended counterpart in Figure 3.7a). Figure 3.7b shows the non-empty definitions for these VPs introduced by the Generic feature, with Figure 3.7c showing the T-New rule with these definitions inlined.

$$\frac{\begin{array}{c}\mathsf{WF_c}(\mathsf{D}, \mathsf{TP_t}\ \mathsf{C}) \\ \mathsf{fields}(\mathsf{TP_t}\ \mathsf{C}) = \overline{\mathsf{V}}\ \overline{\mathsf{f}} \\ \mathsf{D}; \Gamma \vdash \overline{\mathsf{e}} : \overline{\mathsf{U}} \\ \mathsf{D} \vdash \overline{\mathsf{U}} <: \overline{\mathsf{V}}\end{array}}{\mathsf{D}; \Gamma \vdash \mathsf{new}(\, TP_t\ \mathsf{C})(\overline{\mathsf{e}}) : TP_t\ \mathsf{C}} \quad \text{(T-New)}$$

$$\frac{\mathbb{T}}{\mathsf{WF_c}(\epsilon, \mathsf{C}, \epsilon)}$$

$$\mathsf{D} := \epsilon$$

(a)

$$\frac{\Delta \vdash \langle\overline{\mathsf{T}}\rangle\mathsf{C}\ \mathsf{ok}}{\mathsf{WF_c}(\Delta, \langle\overline{\mathsf{T}}\rangle\ \mathsf{C})}$$

$$\mathsf{D} := \Delta$$

(b)

$$\frac{\begin{array}{c}\Delta \vdash \langle\overline{\mathsf{T}}\rangle\mathsf{C}\ \mathsf{ok} \\ \mathsf{fields}(\langle\overline{\mathsf{T}}\rangle\mathsf{C}) = \overline{\mathsf{V}}\ \overline{\mathsf{f}} \\ \Delta; \Gamma \vdash \overline{\mathsf{e}} : \overline{\mathsf{U}} \\ \Delta \vdash \overline{\mathsf{U}} <: \overline{\mathsf{V}}\end{array}}{\Delta; \Gamma \vdash \mathsf{new}(\langle\overline{\mathsf{T}}\rangle\mathsf{C})(\overline{\mathsf{e}}) : \langle\overline{\mathsf{T}}\rangle\mathsf{C}} \quad \text{(GT-New)}$$

(c)

Figure 3.7: Building Generic Typing Rules

3.3.3 Theorem Statements

Variation points also appear in the statements of lemmas and theorems exported by semantic modules, enabling the construction of feature-extensible proofs. Consider the lemma in Figure 3.8 exported by the semantic module cFJ_π which references seven VPs from the syntactic module cFJ_δ. Composing cFJ_δ with modules that export different instantiations of these VPs produce different variations of the productions and rules, with the lemma statement adapting accordingly. Figure 3.9 shows the VP instantiations and the corresponding statement for both cFJ and FGJ (ϵ stands for empty in the cFJ case) with those instantiations inlined for clarity.

$\mathsf{TP_t}$: VP for Class Types $\mathsf{TP_m}$: VP for Method Call Expression μ : VP for Method Types D : Relational Hole for Typing Rules $\mathsf{WF}_{mc}(\mathsf{D}, \mu, \mathsf{TP_m})$: Predicate for T-INVK $\mathsf{WF}_{ne}(\mathsf{D}, \mathsf{TP_t}\ \mathsf{C})$: Predicate for T-NEW $\Phi_M(\mathsf{TP_m}, \mu, \overline{\mathsf{T}})$: Transformation for Return Types
Lemma 3.3.1 (Well-Formed MBody). *If* $\mathsf{mtype}(\mathsf{m}, \mathsf{TP_t}\ \mathsf{C}) = \mu\ \overline{\mathsf{V}} \to \mathsf{V}$*, and with* $\mathsf{WF}_{ne}(\mathsf{D}, TP_t\ \mathsf{C})$ $\mathsf{mbody}(\mathsf{TP_m}, \mathsf{m}, \mathsf{TP_t}\ \mathsf{C}) = \overline{\mathsf{x}}.\mathsf{e}$*, where* $\mathsf{WF}_{mc}(\mathsf{D}, \mu, \mathsf{TP_m})$*, then there exists some* N *and* S *such that* $\mathsf{D} \vdash \mathsf{TP_t}\ \mathsf{C} <: \mathsf{N}$ *and* $\mathsf{D} \vdash \mathsf{S} <: \Phi_M(\mathsf{TP_m}, \mu, \mathsf{V})$ *and* $\mathsf{D}; \overline{\mathsf{x}} : \Phi_M(\mathsf{TP_m}, \mu, \overline{\mathsf{V}}), \mathsf{this} : \mathsf{N} \vdash \mathsf{e} : \mathsf{S}$.

Figure 3.8: VPs in a Parameterized Lemma Statement

$\mathsf{TP_t} : \epsilon$; $\mathsf{TP_m} : \epsilon$; $\mu : \epsilon$; $\mathsf{D} := \epsilon$ $\dfrac{\mathbb{T}}{\mathsf{WF}_{mc}(\epsilon, \epsilon, \overline{\mathsf{T}})}$ $\dfrac{\mathbb{T}}{\mathsf{WF}_{ne}(\epsilon, \mathsf{C})}$ $\Phi_M(\epsilon, \epsilon, \overline{\mathsf{T}}) := \overline{\mathsf{T}}$	**Lemma 3.3.2** (cFJ Well-Formed MBody). *If* $\mathsf{mtype}(\mathsf{m}, \mathsf{C}) = \overline{\mathsf{V}} \to \mathsf{V}$ *and* $\mathbb{T}$ *with* $\mathsf{mbody}(\mathsf{m}, \mathsf{C}) = \overline{\mathsf{x}}.\mathsf{e}$ *where* $\mathbb{T}$*, then there exists some* N *and* S *such that* $\vdash \mathsf{C} <: \mathsf{N}$ *and* $\vdash \mathsf{S} <: \mathsf{V}$ *and* $\overline{\mathsf{x}} : \overline{\mathsf{V}}, \mathsf{this} : \mathsf{N} \vdash \mathsf{e} : \mathsf{S}$.
$\mathsf{TP_t} : \overline{\mathsf{T}}$; $\mathsf{TP_m} : \overline{\mathsf{T}}$; $\mu : \langle \overline{\mathtt{Y} \triangleleft \mathtt{P}} \rangle$; $\mathsf{D} := \Delta$ $\dfrac{\Delta \vdash \langle \overline{\mathsf{T}} \rangle \mathsf{C}\ \mathsf{ok}}{\mathsf{WF}_{ne}(\Delta, \langle \overline{\mathsf{T}} \rangle \mathsf{C})}$ $\dfrac{\Delta \vdash \overline{\mathsf{U}}\ \mathsf{ok} \qquad \Delta \vdash \overline{\mathsf{U}} <: [\overline{\mathsf{U}}/\overline{\mathsf{Y}}]\overline{\mathsf{P}}}{\mathsf{WF}_{mc}(\Delta, \langle \overline{\mathtt{Y} \triangleleft \mathtt{P}} \rangle, \overline{\mathsf{U}})}$ $\Phi_M(\langle \overline{\mathsf{T}} \rangle, \langle \overline{\mathtt{Y} \triangleleft \mathtt{P}} \rangle, \overline{\mathsf{U}}) := [\overline{\mathsf{T}}/\overline{\mathsf{Y}}]\overline{\mathsf{U}}$	**Lemma 3.3.3** (FGJ Well-Formed MBody). *If* $mtype(\mathsf{m}, \langle \overline{\mathsf{T}} \rangle \mathsf{C}) = \langle \overline{\mathsf{Y} \triangleleft \mathsf{P}} \rangle \overline{\mathsf{V}} \to \mathsf{V}$ *and* $\Delta \vdash \langle \overline{\mathsf{T}} \rangle \mathsf{C}\ \mathsf{ok}$ *with* $mbody(\langle \overline{\mathtt{U}} \rangle, \mathsf{m}, \langle \overline{\mathsf{T}} \rangle \mathsf{C}) = \overline{\mathsf{x}}.\mathsf{e}$*, where* $\Delta \vdash \overline{\mathsf{U}}\ \mathsf{ok}$ *and* $\Delta \vdash \overline{\mathsf{U}} <: [\overline{\mathsf{U}}/\overline{\mathsf{Y}}]\overline{\mathsf{P}}$*, then there exists some* N *and* S *such that* $\Delta \vdash \langle \overline{\mathsf{T}} \rangle \mathsf{C} <: \mathsf{N}$ *and* $\Delta \vdash \mathsf{S} <: [\overline{\mathsf{U}}/\overline{\mathsf{Y}}]\mathsf{V}$ *and* $\Delta; \overline{\mathsf{x}} : [\overline{\mathsf{U}}/\overline{\mathsf{Y}}]\overline{\mathsf{V}}, \mathsf{this} : \mathsf{N} \vdash \mathsf{e} : \mathsf{S}$

Figure 3.9: VP Instantiations for cFJ and Generic and the resulting statements of Lemma 4.1 for cFJ and FGJ

Without an accompanying proof, feature-extensible theorem statements are uninteresting. Ideally, a proof should adapt to any variation, allowing it to be reused in any target language variant. This is a fool's errand, as proofs must rule out broken extensions which do not guarantee progress and preservation, and admit only "correct" new cases or VP instantiations.

3.3.4 Crafting Modular Proofs

The goal of semantic modularity is to build reusable metatheory proof pieces which can be composed together alongside syntactic modules, rather than writing multiple related proofs for each language variant. Thus, instead of separately proving the two lemmas in Figure 3.2, the cFJ_π feature exports a proof of the generic Lemma 3.3.4 (Figure 3.10). This lemma is then specialized to the variants FJ and FGJ shown in Figure 3.2 generated by composition of cFJ_δ with Cast_δ and $\mathsf{Cast}_\delta + \mathsf{Generic}_\delta$, respectively. cFJ_π is abstracted over the variation points of cFJ_δ, and can be reused for appropriate instantiations. The proof of Lemma 3.3.4 in cFJ_π reasons over the generic subtyping rules with variation points, as in the case for **S-Dir** in Figure 3.10. The imports of cFJ_δ are opaque within cFJ_π, so this proof will become stuck if it requires knowledge about the behavior of Φ_f or Φ_{SD}.

Lemma 3.3.4. *If* $\Delta \vdash \mathsf{S} <: \mathsf{T}$ *and* $\mathsf{fields}(\Phi_f(\Delta, \mathsf{T})) = \overline{\mathsf{T}}\ \overline{\mathsf{f}}$*, then* $\mathsf{fields}(\Phi_f(\Delta, \mathsf{S})) = \overline{\mathsf{S}}\ \overline{\mathsf{g}}$*,* $\mathsf{S}_i = \mathsf{T}_i$ *and* $\mathsf{g}_i = \mathsf{f}_i$ *for all* $i \leqslant \#(\mathsf{f})$*.*

Case S-Dir
$\mathsf{S} = \mathsf{TP}_0\ \mathsf{C}$, CP_0 $\mathsf{class\ C\ extends\ TP}_1\ \mathsf{D}\ \{\overline{\mathsf{S}}\ \overline{\mathsf{g}}; \ldots\}$,
$\mathsf{T} = \Phi_{SD}(\mathsf{TP}_0, \mathsf{CP}_0, \mathsf{TP}_1\ \mathsf{D})$.
By the rule F-Class, $\mathsf{fields}(\Phi_{SD}(\mathsf{TP}_0, \mathsf{CP}_0, \mathsf{TP}_1\ \mathsf{D})) = \overline{\mathsf{U}}\ \overline{\mathsf{h}}$ with $\mathsf{fields}(\mathsf{TP}_0\ \mathsf{C}) = \overline{\mathsf{U}}\ \overline{\mathsf{h}}; \Phi_{SD}(\mathsf{TP}_0, \mathsf{CP}_0, \overline{\mathsf{S}})\ \overline{\mathsf{g}}$. Assuming that for all class types $\mathsf{TP}_2\ \mathsf{D}'$, $\Phi_f(\Delta, \mathsf{TP}_2\ \mathsf{D}') = \mathsf{TP}_2\ \mathsf{D}'$ and $\Phi_{SD}(\mathsf{TP}_0, \mathsf{CP}_0, \mathsf{TP}_2\ \mathsf{D}')$ returns a class type, $\Phi_f(\Delta, \Phi_{SD}(\mathsf{TP}_0, \mathsf{CP}_0, \mathsf{TP}_1\ \mathsf{D})) = \Phi_{SD}(\mathsf{TP}_0, \mathsf{CP}_0, \mathsf{TP}_1\ \mathsf{D})$. It follows that $\overline{\mathsf{T}}\ \overline{\mathsf{f}} = \mathsf{fields}\Phi_{SD}(\mathsf{TP}_0, \mathsf{CP}_0, \mathsf{TP}_1\ \mathsf{D})) = \overline{\mathsf{U}}\ \overline{\mathsf{h}}$ from which the conclusion is immediate.

Figure 3.10: Generic Statement of Lemmas 3.1.2 and 3.1.1 and Proof for **S-Dir** Case.

In order to proceed, the lemma must constrain possible instantiations of Φ_f or Φ_{SD} to those that have the properties required by the proof. Just as syntactic modules import syntactic definitions, semantic modules import or abstract over *properties* of those definitions in order to export valid proofs. In the case of Lemma 3.3.4, this property is that Φ_f must be the identity function for non-variable types and that Φ_{SD} maps class types to class types. For this proof to hold for the target language, the instantiations of Φ_f and Φ_{SD} must have this property. More concretely, the proof assumes this behavior for all instantiations of Φ_f and Φ_{SD}, producing the new generic Lemma 3.3.5. In order for a feature G_δ which exports Φ_f and Φ_{SD} to be compatible with cFJ_δ, G_π must export the necessary proofs about their behavior. The imports interface of a semantic feature module G_π is the union of all assumptions about the VPs imported by G_δ, while the export interface of G_π is the set of lemmas and theorems the behavior of the VP instantiations and definitions exported by G_δ.

Lemma 3.3.5. *As long as $\Phi_f(\Delta, \mathsf{V}) = \mathsf{V}$ for all non-variable types V and Φ_{SD} maps class types to class types, if $\Delta \vdash \mathsf{S} <: \mathsf{T}$ and $\mathsf{fields}(\Phi_f(\Delta, \mathsf{T})) = \bar{\mathsf{T}}\ \bar{\mathsf{f}}$, then $\mathsf{fields}(\Phi_f(\Delta, \mathsf{S})) = \bar{\mathsf{S}}\ \bar{\mathsf{g}}$, $\mathsf{S}_i = \mathsf{T}_i$ and $\mathsf{g}_i = \mathsf{f}_i$ for all $i \leqslant \#(\mathsf{f})$.*

If a feature adds a new case to a rule or production, a corresponding case must be added to proofs inducting over or case splitting on the original production or rule. For FGJ, this means that a new case must be added to Lemma 5.2 for **GS-Var**. When writing an inductive proof, a semantic feature G_π module provides cases for each of the rules or productions introduced by G_δ. The proof for the union of those cases in the composite module is built by delegating to the module exporting the definition in a given case.

3.4 Looking Forward

The discussion so far has focused on pen-and-paper formalizations. Subsequent chapters consider how to implement these changes in the Coq proof assistant. Modular reasoning within a feature requires a more semantic form of composition that is supported by Coq. OO frameworks are implemented using inheritance and mixin layers [BCS00], techniques that are not available in most proof assistants. Our feature modules instead rely on the higher-order parameterization mechanisms of the Coq theorem prover to support the two kinds of variations discussed. Modules can now be composed within Coq by instantiating parameterized definitions. Using Coq's native abstraction mechanism enables independent certification of each of the feature modules.

Chapter 4

MetaTheory à la Carte: Extensible Datatypes in Coq

We first consider how to add new productions and rules to the syntax and semantics of a programming language embedded in the Coq proof assistant. Syntax and semantics are encoded in Coq as inductive datatypes, with each datatype constructor representing a single production or rule. An expression is a member of the datatype for syntax, while typing and reduction derivations are members of the indexed datatypes for typing and reduction rules, respectively. Figure 4.1 shows the syntax of a simple arithmetic expression language (Arith in BNF, its embedding in pseudo-Coq[1] as an inductive datatype, and the realization of the expression $3 + 2 + 1$ as a member of this datatype.

```
e  ::=
     |   ℕ
     |   e + e
```

```
data Exp_A :=
  | Lit nat
  | Add Exp_A Exp_A

ex_1 :: Exp_A
ex_1 = Add (Lit 3) (Add (Lit 2) (Lit 1))
```

Figure 4.1: Syntax of the Arith expression language.

[1]For clarity of presentation, code in this thesis is written in an adapted version of Coq syntax.

A Logic feature adds syntax for boolean literals and conditional expressions to this expression language. This requires adding new constructors to $\mathsf{Exp_A}$, but defining $\mathsf{Exp_A}$ with **data** has closed it to extension. In order to add new constructors to a type defined by **data**, a completely new definition using **data** is needed, as shown in Figure 4.2. The problem of modularly adding new constructors to datatypes is a manifestation of the well-known "Expression Problem" [Rey94, Coo91, Wad98]. As suggested in the previous chapter, union of constructors requires that non-terminals ($\mathsf{Exp_A}$) be variation points, but the standard inductive datatype mechanism **data** fixes the nonterminal of the type being defined. Thus, supporting case union requires an alternate means for defining datatypes.

$$\begin{array}{rcl} \mathsf{e} & ::= & \mathbb{N} \\ & | & \mathsf{e} + \mathsf{e} \\ & | & \mathbb{B} \\ & | & \textbf{if } \mathsf{e} \textbf{ then } \mathsf{e} \textbf{ else } \mathsf{e} \end{array}$$

$$\begin{array}{l} \textbf{data } \mathsf{Exp_{AB}} := \\ \quad | \ \mathsf{Lit\ nat} \\ \quad | \ \mathsf{Add\ Exp_{AB}\ Exp_{AB}} \\ \quad | \ \mathsf{BLit\ bool} \\ \quad | \ \mathsf{Cond\ Exp_{AB}\ Exp_{AB}\ Exp_{AB}} \end{array}$$

Figure 4.2: Syntax of the $\mathsf{Arith{+}Logic}$ expression language.

A category theoretic presentation of datatypes provides an elegant foundation for the way forward. *Functors* provide a unifying framework for describing datatypes. A functor $\mathsf{F} :: \mathsf{Set} \to \mathsf{Set}$ is simply an operation mapping objects to objects and arrows (functions) to arrows which respects composition:

$$\mathsf{F}(\mathsf{f} \circ \mathsf{g}) = \mathsf{F}(\mathsf{f}) \circ \mathsf{F}(\mathsf{g})$$

and identity:

$$\mathsf{F}(\mathsf{id_A}) = \mathsf{id_{F(A)}}$$

The datatype μ_F "described" by F is the least fixpoint of the functor, or in category theory terms the *initial object* in the category of F-algebras.

This formulation separates describing the *shape* of a datatype using functors from taking the *definition* of datatypes by taking the least fixed point. Coq's datatype definition mechanism combines the two (as do most functional programming languages). In contrast, classes in object-oriented languages such as Java implicitly leave the recursion open in order to support inheritance [Coo89] (although objects suffer from an orthogonal extensibility issue with regards to adding new functions).

Given a operator $\mu :: (\mathsf{Set} \to \mathsf{Set}) \to \mathsf{Set}$ which builds the least fixed point of a functor, the syntax of the Arith expression language is encoded in pseudo-Coq as:

$$\begin{aligned}\mathsf{data\ Arith_F\ e} &= \mathsf{Lit\ n} \mid \mathsf{Add\ e\ e} \\ \mathsf{Exp_A} &= \mu\ \mathsf{Arith_F}\end{aligned}$$

The functor $\mathsf{Arith_F}$ uses its type parameter e for inductive occurrences, leaving the datatype definition open.

Functors can be modified to produce different datatype descriptions. The $\oplus$ operator takes the disjoint sum of two functors:

$$\mathsf{data}\ (\oplus)\ \mathsf{F\ G\ e} = \mathsf{In_L}\ (\mathsf{F\ e}) \mid \mathsf{In_R}\ (\mathsf{G\ e})$$

The $\oplus$ operator builds the functor for $\mathsf{Arith{+}Logic}$ from $\mathsf{Arith_F}$ and the $\mathsf{Logic_F}$ functor describing the constructors of the boolean feature. The μ operator again builds the final datatype for $\mathsf{Arith{+}Logic}$ expressions as the fixpoint of these functors:

$$\begin{aligned}\mathsf{data\ Logic_F\ e} &= \mathsf{BLit\ n} \mid \mathsf{Cond\ e\ e\ e} \\ \mathsf{Exp_{AB}} &= \mu\ (\mathsf{Arith_F} \oplus \mathsf{Logic_F})\end{aligned}$$

Rules for building typing and reduction derivations are embedded as indexed datatypes which can be similarly modularized using *indexed functors*. These functors are parameterized on the type of a index i representing the domain of the relation. Figure 4.3 gives the indexed functor T-Arith describing the typing rules of Arith using the index (i :: Exp $\times$ Ty). The indexed parameter t :: $\forall$(i :: Exp $\times$ Ty), Prop is used for all the subderivations (i.e. $\vdash$ m : natT) in much the same way that e was used in $\mathsf{Arith_F}$.

$$\frac{\mathsf{n} \in \mathbb{N}}{\vdash\ \mathsf{n}\ :\ \mathsf{natT}}\ (\text{T-Lit})$$

$$\frac{\vdash\ \mathsf{m}\ :\ \mathsf{natT} \qquad \vdash\ \mathsf{n}\ :\ \mathsf{natT}}{\vdash\ \mathsf{m} + \mathsf{n}\ :\ \mathsf{natT}}\ (\text{T-Add})$$

```
data T-Arith
  (t :: ∀ (i :: Exp × Ty), Prop) ::
     ∀ (i :: Exp × Ty), Prop:=
  | TLit :: ∀ (n :: nat),
    T-Arith t (Lit n, natT)

  | TAdd :: ∀ (m n :: Exp),
    t m natT → t n natT →
    T-Arith t (Add m n, natT)

data T-Exp_A = μ_i T-Arith
```

Figure 4.3: Typing rules for the Arith expression language.

An indexed variant of μ, μ_i, can be used to build the datatype for typing derivations for the Arith language as the least fixed point of T-Arith. An indexed variant of $\oplus$, $\oplus_i$, builds the disjoint sum of two indexed functor to form the union of two sets of rules:

$$\mathsf{data}\ (\oplus_\mathsf{i})\ \mathsf{F_i}\ \mathsf{G_i}\ \mathsf{e_i}\ =\ \mathsf{In_L}\ (\mathsf{F_i}\ \mathsf{e_i})\ |\ \mathsf{In_R}\ (\mathsf{G_i}\ \mathsf{e_i})$$

Computing with Functors

Functions are typically defined by case analysis over an inductive datatype, as in the evaluation function for Arith in Figure 4.4. The switch to defining datatypes as a fixpoint of a

$$\frac{}{[\![\mathsf{n}]\!] \rightsquigarrow \mathsf{n}}$$

$$\frac{[\![\mathsf{e_m}]\!] \rightsquigarrow \mathsf{m} \qquad [\![\mathsf{e_n}]\!] \rightsquigarrow \mathsf{n}}{[\![\mathsf{e_m} + \mathsf{e_n}]\!] \rightsquigarrow \mathsf{m} + \mathsf{n}}$$

```
Eval-Arith :: Exp_A → Val
Eval-Arith (Lit n) = n

Eval-Arith (Add m n) = (Eval-Arith m) + (Eval-Arith n)
```

Figure 4.4: Evaluation function for the Arith expression language.

functor using the μ operator requires a different method of function definition. Functions from the fixpoint of a functor F to a type A can be built as folds of *F-algebras*, or morphisms from $\mathsf{F(A)}$ to A: $\mathsf{f} :: \mathsf{F(A)} \to \mathsf{A}$. Since μ_F is the initial object in the category of F-algebras, for each F-algebra f there exists a unique function, $\mathsf{fold_f} : \mu_\mathsf{F} \to \mathsf{A}$, such that:

$$\mathsf{fold_f} \circ \mathsf{in} = \mathsf{f} \circ \mathsf{F}(\mathsf{fold_f}) \tag{4.1}$$

Expressed as a commuting diagram:

$$\begin{tikzcd}
\mathsf{F}(\mu_\mathsf{F}) \arrow[r, "\mathsf{in}"] \arrow[d, "\mathsf{F(fold\ f)}"'] & \mu_\mathsf{F} \arrow[d, "\mathsf{fold\ f}"] \\
\mathsf{F(A)} \arrow[r, "f"'] & \mathsf{A}
\end{tikzcd}$$

Functions are thus defined as F-algebras over a functor F:

```
type Algebra f a  =  f a → a

Eval-Arith :: Algebra Arith_F Value

Eval-Arith (Lit n)                            =   NValue n

Eval-Arith (Add (NValue m) (NValue n))        =   NValue (m + n)

Eval_A                                        =   fold Eval-Arith
```

Another composition operator, $\boxplus$, builds $\mathsf{F} \oplus \mathsf{G}$-algebras for composite functors:

```
(⊞)  ::  Algebra F V  →  Algebra G V  →  Algebra (F ⊕ G) V

alg_F  ⊞  alg_G (In_L e)                                   =   alg_F e

alg_F  ⊞  alg_G (In_R e)                                   =   alg_G e
```

Given an evaluation algebra $\mathsf{Eval\text{-}Logic}$ for the $\mathsf{Logic_F}$ functor, $\boxplus$ can build an interpreter for $\mathsf{Exp_{AB}}$:

$$\mathsf{Eval_{AB}} \quad = \quad \mathsf{fold}\ (\mathsf{Eval\text{-}Arith} \boxplus \mathsf{Eval\text{-}Logic})$$

4.1 Extensible Datatypes in MTC

The techniques described so far form the foundation for the *Data Types à la Carte* (DTC) [Swi08] solution to the Expression Problem. In DTC, the explicitly recursive definition Fix f closes the open recursion of a functor F.

data Fix F = In (F (Fix F))

fold is similarly expressed using general recursion.

fold :: Functor f ⇒ Algebra f a → Fix f → a

fold alg (In fa) = alg (fmap (fold alg) fa)

Unfortunately, these two uses of general recursion are not permitted in Coq. Coq does not accept the type-level fixpoint combinator Fix f because it is not strictly positive. Coq similarly disallows the fold function because it is not structurally recursive. The remainder of this chapter details alternate definitions of these two functions which can be implemented in Coq. This solution is realized as the *Metatheory ̀la Carte* (MTC) framework for extensible datatypes in Coq.

4.1.1 Recursion-Free Church Encodings

MTC encodes data types and folds with Church encodings [BB85, PPM90], which are recursion-free. Church encodings represent (least) fixpoints and folds as follows:

type Fix f = ∀a.Algebra f a → a

$\mathsf{fold :: Algebra\ f\ a \rightarrow Fix\ f \rightarrow a}$

$\mathsf{fold\ alg\ fa = fa\ alg}$

Both definitions are non-recursive and can be encoded in Coq (although we need to enable impredicativity for certain definitions). Since Church encodings represent data types as folds, the definition of fold is trivial: it applies the folded Fix f data type to the algebra.

Example Church encodings of $\mathsf{Arith_F}$'s literals and addition are given by the lit and add functions:

$\mathsf{lit :: Nat \rightarrow Fix\ Arith_F}$

$\mathsf{lit\ n = \lambda alg \rightarrow alg\ (Lit\ n)}$

$\mathsf{add :: Fix\ Arith_F \rightarrow Fix\ Arith_F \rightarrow Fix\ Arith_F}$

$\mathsf{add\ e_1\ e_2 = \lambda alg \rightarrow alg\ (Add\ (fold\ alg\ e_1)\ (fold\ alg\ e_2))}$

Note once again that the definitions of Lit and Add in the $\mathsf{Arith_F}$ functor are both recursion-free. The church-encodings of the two constructors take care of the recursion instead, applying fold alg to each of the subterms.

Evaluation algebras and interpreters for $\mathtt{Exp_A}$ can be defined as in DTC, and expressions are evaluated in the same way.

4.1.2 Lack of Control over Recursion

Folds are inherently structurally recursive, as algebras assume that all subterms have been recursively evaluated. This denies the implementer of the algebra explicit control over recursion. Church-encoding based on folds reduce all subterms; the only freedom in the algebra is whether or not to use the recursively-obtained result. This is obviously not ideal for all functions.

Example: Modeling if expressions As a simple example that illustrates the issue of lack of control over recursive application, consider modeling the big-step semantics of **if**

expressions:

$$\frac{[\![\mathsf{e}_1]\!] \rightsquigarrow \textbf{true} \qquad [\![\mathsf{e}_2]\!] \rightsquigarrow \mathsf{v}_2}{[\![\,\textbf{if}\ \mathsf{e}_1\ \mathsf{e}_2\ \mathsf{e}_3]\!] \rightsquigarrow \mathsf{v}_2} \qquad\qquad \frac{[\![\mathsf{e}_1]\!] \rightsquigarrow \textbf{false} \qquad [\![\mathsf{e}_3]\!] \rightsquigarrow \mathsf{v}_3}{[\![\,\textbf{if}\ \mathsf{e}_1\ \mathsf{e}_2\ \mathsf{e}_3]\!] \rightsquigarrow \mathsf{v}_3}$$

Evaluation for the boolean feature is defined as an algebra over the $\mathsf{Logic_F}$ functor:

$$\begin{aligned}
&\mathsf{eval_{Logic}} :: \mathsf{Algebra\ Logic_F\ Value}\\
&\mathsf{eval_{Logic}}\ (\mathsf{If}\ \mathsf{v}_1\ \mathsf{v}_2\ \mathsf{v}_3) = \textbf{if}\ (\mathsf{v}_1 \equiv \mathsf{BVal\ True})\ \textbf{then}\ \mathsf{v}_2\ \textbf{else}\ \mathsf{v}_3\\
&\mathsf{eval_{Logic}}\ (\mathsf{BLit\ b}) \qquad = \mathsf{B\ b}
\end{aligned}$$

However, an important difference with the big-step semantics above is that $\mathsf{eval_{Logic}}$ cannot control where evaluation happens. All it has in hand are the values v_1, v_2 and v_3 that result from evaluation. While this difference is not important for simple features like arithmetic expressions, it does matter for **if** expressions.

4.1.3 Mendler-style Church Encodings

To express functions in a way that allows explicit control over recursion, MTC adapts Church encodings to use Mendler-style algebras and folds [UV00] which make recursive calls explicit.

$$\textbf{type}\ \mathsf{MAlgebra\ F\ a} = \forall \mathsf{r}.(\mathsf{r} \to \mathsf{a}) \to \mathsf{f\ r} \to \mathsf{a}$$

A Mendler-style algebra differs from a traditional F-algebra in that it takes an additional argument $(\mathsf{r} \to \mathsf{a})$ which corresponds to recursive calls. To ensure that recursive calls can only be applied structurally, the arguments that appear at recursive positions have a polymorphic type r. The use of this polymorphic type r prevents case analysis, or any other type of inspection, on those arguments. Using $\mathsf{MAlgebra\ f\ a}$, Mendler-style folds and Mendler-style Church encodings are defined as follows:

$$\textbf{type}\ \mathsf{Fix_M\ f} = \forall \mathsf{a}.\mathsf{MAlgebra\ f\ a} \to \mathsf{a}$$

$\mathsf{fold_M} :: \mathsf{MAlgebra\ f\ a} \rightarrow \mathsf{Fix_M\ f} \rightarrow \mathsf{a}$

$\mathsf{fold_M\ alg\ fa} = \mathsf{fa\ alg}$

Mendler-style folds allow algebras to state their recursive calls explicitly. As an example, the definition of the evaluation of **if** expressions in terms of a Mendler-style algebra is:

$\mathsf{eval_{Logic}} :: \mathsf{MAlgebra\ Logic_F\ Value}$

$\mathsf{eval_{Logic}}\ [\![\cdot]\!]\ (\mathsf{BLit\ b}) \qquad = \mathsf{B\ b}$

$\mathsf{eval_{Logic}}\ [\![\cdot]\!]\ (\mathsf{If\ e_1\ e_2\ e_3}) = \textbf{if}\ ([\![\mathsf{e_1}]\!] \equiv \mathsf{B\ True})\ \textbf{then}\ [\![\mathsf{e_2}]\!]$

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ \ [\![\mathsf{e_3}]\!]$

Note that this definition allows explicit control over the evaluation order just like the big-step semantics definition. Furthermore, like the fold-definition, $\mathsf{eval_{Logic}}$ enforces compositionality because all the algebra can do to $\mathsf{e_1}$, $\mathsf{e_2}$ or $\mathsf{e_3}$ is to apply the recursive call $[\![\cdot]\!]$.

4.1.4 A Compositional Framework for Mendler-style Algebras

DTC provides a convenient framework for composing conventional fold algebras. MTC provides a similar framework, but for Mendler-style algebras instead of F-algebras. In order to write modular proofs, MTC regulates its definitions with a number of laws.

Modular Mendler Algebras A type class is defined for every extensible function. Given an extensible datatype for values, Value, the evaluation function has the following class:

$\textbf{class}\ \mathsf{Eval\ f}\ \textbf{where}\ \mathsf{eval_{alg}} :: \mathsf{MAlgebra\ f\ Value}$

In this class $\mathsf{eval_{alg}}$ represents the evaluation algebra of a feature f.
Algebras for composite functors are built from feature algebras[2]:

$\textbf{instance}\ (\mathsf{Eval\ f}, \mathsf{Eval\ g}) \Rightarrow \mathsf{Eval\ (f \oplus g)}\ \textbf{where}$

$\quad \mathsf{eval_{alg}}\ [\![\cdot]\!]\ (\mathsf{Inl\ fexp})\ = \mathsf{eval_{alg}}\ [\![\cdot]\!]\ \mathsf{fexp}$

$\quad \mathsf{eval_{alg}}\ [\![\cdot]\!]\ (\mathsf{Inr\ gexp}) = \mathsf{eval_{alg}}\ [\![\cdot]\!]\ \mathsf{gexp}$

[2]This instance corresponds to the ⊞ operator described previously.

```
class f ≺: g where
  inj       :: f a → g a
  prj       :: g a → Maybe (f a)
  inj_prj :: prj ga = Just fa → ga = inj fa   -- law
  prj_inj :: prj ∘ inj = Just                 -- law
instance (f ≺: g) ⇒ f ≺: (g ⊕ h) where
  inj fa        = Inl (inj fa)
  prj (Inl ga) = prj ga
  prj (Inr ha) = Nothing
instance (f ≺: h) ⇒ f ≺: (g ⊕ h) where
  inj fa        = Inr (inj fa)
  prj (Inl ga) = Nothing
  prj (Inr ha) = prj ha
instance f ≺: f where
  inj fa = fa
  prj fa = Just fa
```

Figure 4.5: Functor subtyping.

Overall evaluation can then be defined as:

$$\mathsf{eval} :: \mathsf{Eval}\ \mathsf{f} \Rightarrow \mathsf{Fix_M}\ \mathsf{f} \rightarrow \mathsf{Value}$$

$$\mathsf{eval} = \mathsf{fold_M}\ \mathsf{eval_{alg}}$$

In order to avoid the repeated boilerplate of defining a new type class for every semantic function and corresponding instance for ⊕, MTC defines a single generic Coq type class, FAlg, that is indexed by the name of the semantic function. This class definition can be found in Figure 4.7 and subsumes all other algebra classes found in this chapter. The chapter continues to use more specific classes to make a gentler progression for the reader.

Injections and Projections of Functors Figure 4.5 shows the multi-parameter type class ≺:. This class provides a means to lift or inject (inj) (sub)functors f into larger compositions g and project (prj) them out again. The inj_prj and prj_inj laws relate the injection and projection methods in the ≺: class, ensuring that the two are effectively inverses. The idea is to use the type class resolution mechanism to encode (coercive) subtyping between functors. In Coq this subtyping relation can be nicely expressed because Coq type classes [SO08] perform a backtracking search for matching instances. Thus, highly overlapping definitions like the first and second instances are allowed. This is a notable difference to Haskell's type classes, which do not support backtracking. Hence, DTC's Haskell solution has to provide a biased choice that does not accurately model the expected subtyping relationship.

The $\mathsf{in_f}$ function builds a new term from the application of f to some subterms.

$\mathsf{in_f :: f\ (Fix_M\ f) \to Fix_M\ f}$
$\mathsf{in_f\ fexp = \lambda alg \to alg\ (fold_M\ alg)\ fexp}$

Smart constructors automatically inject terms into a supertype using $\mathsf{in_f}$ and inj:

$\mathsf{inject :: (g \prec: f) \Rightarrow g\ (Fix_M\ f) \to Fix_M\ f}$
$\mathsf{inject\ gexp = in_f\ (inj\ gexp)}$

$\mathsf{lit :: (Arith_F \prec: f) \Rightarrow Nat \to Fix_M\ f}$
$\mathsf{lit\ n = inject\ (Lit\ n)}$

$\mathsf{blit :: (Logic_F \prec: f) \Rightarrow Bool \to Fix_M\ f}$
$\mathsf{blit\ b = inject\ (BLit\ b)}$

$\mathsf{cond :: (Logic_F \prec: f)}$
$\quad\mathsf{\Rightarrow Fix_M\ f \to Fix_M\ f \to Fix_M\ f \to Fix_M\ f}$
$\mathsf{cond\ c\ e_1\ e_2 = inject\ (If\ c\ e_1\ e_2)}$

Expressions are built with the smart constructors and used by operations like evaluation:

$\mathsf{exp :: Fix_M\ (Arith_F \oplus Logic_F)}$
$\mathsf{exp = cond\ (blit\ True)\ (lit\ 3)\ (lit\ 2)}$

```
> eval exp
3
```

The $\mathsf{out_f}$ function exposes the toplevel functor again:

```
out_f :: Functor f ⇒ Fix_M f → f (Fix_M f)
out_f exp = fold_M (λrec fr → fmap (in_f ∘ rec) fr) exp
```

We can pattern match on particular features using prj and $\mathsf{out_f}$:

```
project :: (g ≺: f, Functor f) ⇒
  Fix_M f → Maybe (g (Fix_M f))
project exp = prj (out_f exp)
isLit :: (Arith_F ≺: f, Functor f) ⇒ Fix_M f → Maybe Nat
isLit exp = case project exp of
  Just (Lit n) → Just n
  Nothing      → Nothing
```

4.1.5 Extensible Semantic Values

In addition to modular language features, it is also desirable to have modular result types for semantic functions. For example, it is much cleaner to separate natural number and boolean values along the same lines as the $\mathsf{Arith_F}$ and $\mathsf{Logic_F}$ features. To easily achieve this extensibility, we make use of the same sorts of extensible encodings as the expression language itself:

```
data NVal_F a  = I Nat
data BVal_F a  = B Bool
data Stuck_F a = Stuck
vi :: (NVal_F ≺: r) ⇒ Nat → Fix_M r
vi n = inject (I n)

vb :: (BVal_F ≺: r) ⇒ Bool → Fix_M r
vb b = inject (B b)

stuck :: (Stuck_F ≺: r) ⇒ Fix_M r
stuck = inject Stuck
```

Besides constructors for integer (vi) and boolean (vb) values, we also include a constructor denoting stuck evaluation (stuck).

To allow for an extensible return type r for evaluation, we need to parametrize the Eval type class in r:

```
class Eval f r where
  eval_alg :: MAlgebra f (Fix_M r)
```

Projection is now essential for pattern matching on values:

```
instance (Stuck_F ≺: r, NVal_F ≺: r, Functor r) ⇒
  Eval Arith_F r where
    eval_alg ⟦·⟧ (Lit n)       = vi n
    eval_alg ⟦·⟧ (Add e1 e2) =
      case (project ⟦e1⟧, project ⟦e2⟧) of
        (Just (I n1), (Just (I n2))) → vi (n1 + n2)
        _                            → stuck
```

This concludes MTC's support for extensible inductive data types and functions. To cater to meta-theory, MTC must also support reasoning about these modular definitions.

4.2 Reasoning with Church Encodings

While Church encodings are the foundation of extensibility in MTC, Coq does not provide induction principles for them. It is an open problem to do so without resorting to axioms.

MTC solves this problem with a novel axiom-free approach based on adaptations of two important aspects of folds discussed by Hutton [Hut99].

4.2.1 The Problem of Church Encodings and Induction

Coq's own original approach [PPM90] to inductive data types was based on Church encodings. It is well-known that Church encodings of inductive data types have problems expressing induction principles such as $\mathsf{A}_{\mathsf{ind}}$, the induction principle for arithmetic expressions.

```
A_ind :: ∀P :: (Arith → Prop).
        ∀H_l :: (∀n.P (Lit n)).
        ∀H_a :: (∀a b.P a → P b → P (Add a b)).
        ∀a. P a
A_ind P H_l H_a e =
  case e of Lit n → H_l n
    Add x y      → H_a a b (A_ind P H_l H_a x)
                           (A_ind P H_l H_a y)
```

The original solution to this problem in Coq involved axioms for induction, which endangered strong normalization of the calculus (among other problems). This was the primary motivation for the creation of the *Calculus of Inductive Constructions* [PM93] which extended the Calculus of Constructions with built-in inductive data types.

Why exactly are proofs problematic for Church encodings, where inductive functions are not? After all, a Coq proof is essentially a function that builds a proof term by induction over a data type, so the Church encoding should be able to express a proof as a fold with a *proof algebra* over the data type, in the same way it represents other functions.

The problem is that this approach severely restricts the propositions that can be proven. Folds over Church encodings are destructive, so their result type cannot depend on the term being destructed. For example, it is impossible to express the proof for *type soundness* because it performs induction over the expression e mentioned in the type soundness property.

$$\forall \mathsf{e}.\ \Gamma \vdash \mathsf{e} : \mathsf{t} \rightarrow \Gamma \vdash [\![\mathsf{e}]\!] : \mathsf{t}$$

This restriction is a showstopper for programming language metatheory, as it rules out proofs for most (if not all) theorems of interest.

4.2.2 Type Dependency with Dependent Products

Hutton's first aspect of folds is that they become substantially more expressive with the help of tuples. The dependent products in Coq take this observation one step further. While an f-algebra cannot refer to the original term, it can simultaneously build a copy e of the original term and a proof that the property $\mathsf{P}\ \mathsf{e}$ holds for the new term. As the latter depends on the former, the result type of the algebra is a dependent product $\Sigma\ \mathsf{e}.\mathsf{P}\ \mathsf{e}$. A generic algebra can exploit this expressivity to build a poor-man's induction principle, e.g., for the $\mathsf{Arith_F}$ functor:

```
A²ind :: ∀P :: (Fix_M Arith_F → Prop).
      ∀H_l :: (∀n.P (lit n)).
      ∀H_a :: (∀a b.P a → P b → P (add a b)).
      Algebra Arith_F (Σ e.P e)
A²ind P H_l H_a e =
  case e of
    Lit n    → ∃ (lit n) (H_l n)
    Add x y → ∃ (add (π1 x) (π1 y)) (H_a (π1 x) (π1 y)
                                        (π2 x) (π2 y))
```

Provided with the necessary proof cases, $\mathsf{A}^2_{\mathsf{ind}}$ can build a specific proof algebra. The corresponding proof is simply a fold over a Church encoding using this proof algebra.

Since the efficiency of a proof is not usually a concern, it makes more sense to use regular algebras than Mendler algebras. Fortunately, regular algebras are compatible with Mendler-based Church encodings as the following variant of $\mathsf{fold}'_{\mathsf{M}}$ shows.

$$\mathsf{fold}'_{\mathsf{M}} :: \mathsf{Functor\ f} \Rightarrow \mathsf{Algebra\ f\ a} \rightarrow \mathsf{Fix}_{\mathsf{M}}\ \mathsf{f} \rightarrow \mathsf{a}$$

$$\mathsf{fold}'_{\mathsf{M}}\ \mathsf{alg} = \mathsf{fold}_{\mathsf{M}}\ (\lambda \mathsf{rec} \rightarrow \mathsf{alg} \circ \mathsf{fmap\ rec})$$

4.2.3 Term Equality with the Universal Property

Of course, the dependent product approach does not directly prove a property of the original term. Instead, given a term, it builds a new term and a proof that the property holds for the new term. In order to draw conclusions about the original term from the result, the original and new term must be equal.

Clearly the equivalence does not hold for arbitrary terms that happen to match the type signatures $\mathsf{Fix}_{\mathsf{M}}\ \mathsf{f}$ for Church encodings and $\mathsf{Algebra\ f}\ (\Sigma\ \mathsf{e}.\mathsf{P\ e})$ for proof algebras. Statically ensuring this equivalence requires additional *well-formedness conditions* on both. These conditions formally capture our notion of "correct" Church encodings and proof algebras.

Well-Formed Proof Algebras

The first requirement, for algebras, states that the new term produced by application of the algebra is equal to the original term.

$$\forall \mathsf{alg} :: \mathsf{Algebra\ f}\ (\Sigma\ \mathsf{e}.\mathsf{P\ e}).\ \pi_1\ \circ\ \mathsf{alg} = \mathsf{in}_{\mathsf{f}}.\mathsf{fmap}\ \pi_1$$

This constraint is encoded in the typeclass for proof algebras, PAlg. It is easy to verify that $\mathsf{A}^2_{\mathsf{ind}}$ satisfies this property. Other proof algebras over $\mathsf{Arith}_{\mathsf{F}}$ can be defined by instantiating $\mathsf{A}^2_{\mathsf{ind}}$ with appropriate cases for H_{l} and H_{a}. In general, well-formedness needs to be proven only once for any data type and induction algebra.

Well-Formed Church Encodings

Well-formedness of proof algebras is not enough because a proof is not a single application of an algebra, but rather a $\mathsf{fold}'_{\mathsf{M}}$ of it. So the $\mathsf{fold}'_{\mathsf{M}}$ used to build a proof must be a *proper* $\mathsf{fold}'_{\mathsf{M}}$. As the Church encodings represent inductive data types as their folds, this boils down to ensuring that the Church encodings are well-formed.

Hutton's second aspect of folds formally characterizes the definition of a fold using its *universal property*:

$$\mathsf{h} = \mathsf{fold}'_{\mathsf{M}}\ \mathsf{alg} \Leftrightarrow \mathsf{h} \circ \mathsf{in}_{\mathsf{f}} = \mathsf{alg}\ \mathsf{h}$$

In an initial algebra representation of an inductive data type, there is a single implementation of $\mathsf{fold}'_{\mathsf{M}}$ that can be checked once and for all for the universal property. In MTC's Church-encoding approach, every term of type $\mathsf{Fix}_{\mathsf{M}}\ \mathsf{f}$ consists of a separate $\mathsf{fold}'_{\mathsf{M}}$ implementation that must satisfy the universal property. Note that this definition of the universal property is for a $\mathsf{fold}'_{\mathsf{M}}$ using a traditional algebra. As the only concern is the behavior of proof algebras (which are traditional algebras) folded over Church encodings, this is a sufficient characterization of well-formedness. Hinze [Hin05] uses the same characterization for deriving Church numerals.

Fortunately, the left-to-right implication follows trivially from the definitions of $\mathsf{fold}'_{\mathsf{M}}$ and in_{f}, independent of the particular term of type $\mathsf{Fix}_{\mathsf{M}}\ \mathsf{f}$. Thus, the only hard well-formedness requirement for a Church-encoded term e is that it satisfies the right-to-left implication of the universal property.

```
type UP f e =
  ∀a (alg :: Algebra_M f a) (h :: Fix_M f → a).
    (∀e′. h (in_f e′) = alg h e′) → h e = fold′_M alg e
```

This property is easy to show for any given smart constructor. MTC actually goes one step further and redefines its smart constructors in terms of a new in_{f}, that only builds terms with the universal property

$\mathsf{in}'_{\mathsf{f}} :: \mathsf{Functor\ f} \Rightarrow \mathsf{f}\ (\Sigma\ \mathsf{e.UP\ f\ e}) \rightarrow \Sigma\ \mathsf{e.UP\ f\ e}$

about Church-encoded terms built from these smart-*er* constructors, as all of the nice properties of initial algebras hold for these terms and, importantly, these properties provide a handle on reasoning about these terms.

Two known consequences of the universal property are the famous *fusion* law, which describes the composition of a fold with another computation,

$$\mathsf{h} \circ \mathsf{alg}_1 = \mathsf{alg}_2 \circ \mathsf{fmap\ h} \quad \Rightarrow \quad \mathsf{h} \circ \mathsf{fold}'_{\mathsf{M}}\ \mathsf{alg}_1 = \mathsf{fold}'_{\mathsf{M}}\ \mathsf{alg}_2$$

and the lesser known *reflection* law,

$$\mathsf{fold}'_{\mathsf{M}}\ \mathsf{in}_{\mathsf{f}} = \mathsf{id}$$

Soundness of Input-Preserving Folds

Armed with the two well-formedness properties, we can prove the key theorem for building inductive proofs over Church encodings:

Theorem 4.2.1. *Given a functor* f*, property* P*, and a well-formed* P*-proof algebra* alg*, for any Church-encoded* f*-term* e *with the universal property, we can conclude that* $\mathsf{P\ e}$ *holds.*

Proof. Given that $\mathsf{fold}'_{\mathsf{M}}\ \mathsf{alg\ e}$ has type $\Sigma\ \mathsf{e}'.\mathsf{P\ e}'$, we have that $\pi_2\ (\mathsf{fold}'_{\mathsf{M}}\ \mathsf{alg\ e})$ is a proof for $\mathsf{P}\ (\pi_1\ (\mathsf{fold}'_{\mathsf{M}}\ \mathsf{alg\ e}))$. From that the lemma is derived as follows:

$\mathsf{P}\ (\pi_1\ (\mathsf{fold}'_{\mathsf{M}}\ \mathsf{alg\ e}))$

$\Longrightarrow$ {-well-founded algebra and fusion law -}

$\mathsf{P}\ (\mathsf{fold}'_{\mathsf{M}}\ \mathsf{in}_{\mathsf{f}}\ \mathsf{e})$

$\Longleftrightarrow$ {-reflection law -}

$\mathsf{P\ e}$ $\square$

Theorem 4.2.1 enables the construction of a statically-checked proof of correctness as an input-preserving fold of a proof algebra. This provides a means to achieve our true goal: modular proofs for extensible Church encodings.

4.3 Modular Proofs for Extensible Church Encodings

The goal of modularity is to write separate proofs for every feature which can be composed together into an overall proof for the feature composition. These proofs should be independent from one another, so that they can be reused for different combinations of features.

Fortunately (and deliberately), since proofs are essentially folds of proof algebras, all of the reuse tools developed in Section 4.1 apply here. In particular, composing proofs is a simple matter of combining proof algebras with $\oplus$. Nevertheless, the transition to modular components does introduce several wrinkles in the reasoning process.

4.3.1 Algebra Delegation

Due to injection, propositions range over the abstract (super)functor f of the component composition. The signature of $\mathsf{A}^2_{\mathsf{ind}}$, for example, becomes:

$$\begin{array}{l} \mathsf{A}^2_{\mathsf{ind}} :: \forall \mathsf{f}.\ \mathsf{Arith}_\mathsf{F} \prec: \mathsf{f} \Rightarrow \\ \quad \forall \mathsf{P} :: (\mathsf{Fix}_\mathsf{M}\ \mathsf{f} \rightarrow \mathsf{Prop}). \\ \quad \forall \mathsf{H}_\mathsf{l} :: (\forall \mathsf{n}.\mathsf{P}\ (\mathsf{lit}\ \mathsf{n})). \\ \quad \forall \mathsf{H}_\mathsf{a} :: (\forall \mathsf{a}\ \mathsf{b}.\mathsf{P}\ \mathsf{a} \rightarrow \mathsf{P}\ \mathsf{b} \rightarrow \mathsf{P}\ (\mathsf{add}\ \mathsf{a}\ \mathsf{b})). \\ \quad \mathsf{Algebra}\ \mathsf{Arith}_\mathsf{F}\ (\Sigma\ \mathsf{e}.\mathsf{P}\ \mathsf{e}) \end{array}$$

Consider building a proof of

$$\forall \mathsf{e}.\ \mathsf{typeof}\ \mathsf{e} = \mathsf{Just}\ \mathsf{nat} \rightarrow \exists\ \mathsf{m} :: \mathsf{nat}.\ \mathsf{eval}\ \mathsf{e} = \mathsf{vi}\ \mathsf{m}$$

using $\mathsf{A}^2_{\mathsf{ind}}$. Then, the first proof obligation is

$\textbf{instance}\ (\mathsf{Eval\ f}, \mathsf{Eval\ g}, \mathsf{Eval\ h}, \mathsf{WF\_Eval\ f\ g}) \Rightarrow$
$\quad \mathsf{WF\_Eval\ (f} \prec: \mathsf{g} \oplus \mathsf{h)}$
$\textbf{instance}\ (\mathsf{Eval\ f}, \mathsf{Eval\ g}, \mathsf{Eval\ h}, \mathsf{WF\_Eval\ f\ h}) \Rightarrow$
$\quad \mathsf{WF\_Eval\ (f} \prec: \mathsf{g} \oplus \mathsf{h)}$
$\textbf{instance}\ (\mathsf{Eval\ f}) \Rightarrow \mathsf{WF\_Eval\ f\ f}$

Figure 4.6: WF_Eval instances.

$\mathsf{typeof\ (lit\ n)} = \mathsf{Just\ nat} \rightarrow \exists\ \mathsf{m :: nat.\ eval\ (lit\ n)} = \mathsf{vi\ m}$

While this appears to follow immediately from the definition of eval, recall that eval is a fold of an abstract algebra over f and is thus opaque. To proceed, we need the additional property that this f-algebra delegates to the $\mathsf{Arith_F}$-algebra as expected:

$\forall \mathsf{r\ (rec :: r} \rightarrow \mathsf{Nat).\ eval_{alg}\ rec.inj} = \mathsf{eval_{alg}\ rec}$

This delegation behavior follows from our approach: the intended structure of f is a $\oplus$-composition of features, and $\oplus$-algebras are intended to delegate to the feature algebras. We can formally capture the delegation behavior in a type class that serves as a precondition in our modular proofs.

$\textbf{class}\ (\mathsf{Eval\ f}, \mathsf{Eval\ g}, \mathsf{f} \prec: \mathsf{g}) \Rightarrow$
$\quad \mathsf{WF\_Eval\ f\ g}\ \textbf{where}$
$\quad \mathsf{wf\_eval\_alg} :: \forall \mathsf{r\ (rec :: r} \rightarrow \mathsf{Nat)\ (e :: f\ r)}.$
$\qquad \mathsf{eval_{alg}\ rec\ (inj\ e :: g\ r)} =$
$\qquad \mathsf{eval_{alg}\ rec\ e}$

MTC provides the three instances of this class in Figure 4.6, one for each instance of $\prec:$, allowing Coq to automatically build a proof of well-formedness for every composite algebra.

Automating Composition

A similar approach is used to automatically build the definitions and proofs of languages from pieces defined by individual features. In addition to functor and algebra composition, the framework derives several important reasoning principles as type class instances, similarly to WF_Eval. These include the DistinctSubFunctor class, which ensures that injections from two different subfunctors are distinct, and the WF_Functor class that ensures that fmap distributes through injection.

Figure 4.7 provides a summary of all the classes defined in MTC, noting whether the base instances of a particular class are provided by the user or inferred with a default instance. Importantly, instances of all these classes for feature compositions are built automatically, analogously to the instances in Figure 4.6.

4.3.2 Extensible Inductive Predicates

Many proofs appeal to rules which define a predicate for an important property. In Coq these predicates are expressed as inductive data types of kind Prop. For instance, a soundness proof makes use of a judgment about the well-typing of values.

```
data WTValue :: Value → Type → Prop where
  WTNat  :: ∀n. WTValue (I n) TNat
  WTBool :: ∀b. WTValue (B b) TBool
```

When dealing with a predicate over extensible inductive data types, the set of rules defining the predicate must be extensible as well. Extensibility of these rules is obtained in much the same way as that of inductive data types: by means of Church encodings. The important difference is that logical relations are indexed data types: e.g., WTValue is indexed by a value and a type. This requires functors indexed by values x of type i. For example, $\mathsf{WTNat}_{\mathsf{F}}$ v t is the corresponding indexed functor for the extensible variant of WTNat above.

Class Definition	Description
class Functor f **where** fmap :: (a → b) → (f a → f b) fmap_id :: fmap id = id fmap_fusion :: ∀g h. fmap h ∘ fmap g = fmap (h ∘ g)	**Functors** Supplied by the user
class f ≺: g **where** inj :: f a → g a prj :: g a → Maybe (f a) inj_prj :: prj ga = Just fa → ga = inj fa prj_inj :: prj ∘ inj = Just	Functor Subtyping Inferred
class (Functor f, Functor g, f ≺: g) ⇒ WF_Functor f g **where** wf_functor :: ∀a b (h :: a → b). fmap h ∘ inj = inj ∘ fmap h	Functor Delegation Inferred
class (Functor h, f ≺: h, g ≺: h) ⇒ DistinctSubFunctor f g h **where** inj_discriminate :: ∀a (fe :: f a) (ge :: g a).inj fe ≠ inj ge	Functor Discrimination Inferred
class FAlg name t a f **where** f_algebra : Mixin t f a	**Function Algebras** Supplied by the user
class (f ≺: g, FAlg n t a f, FAlg n t a g) ⇒ WF_FAlg n t a f g **where** wf_algebra :: ∀rec (fa :: f t). f_algebra rec (inj fa) = f_algebra rec fa	Algebra Delegation Inferred
class (Functor f, Functor g, f ≺: g) ⇒ PAlg name f g a **where** p_algebra :: Algebra f a proj_eq :: ∀e.π_1 (p_algebra e) = in_f (inj (fmap π_1 e))	**Proof Algebras** Supplied by the User

Figure 4.7: Type classes provided by MTC

$$
\begin{aligned}
&\textbf{data}\ \mathsf{WTNat_F} :: \mathsf{v} \to \mathsf{t} \to (\mathsf{WTV}\ ::\ (\mathsf{v},\mathsf{t}) \to \mathsf{Prop})\\
&\qquad\qquad\qquad\qquad\qquad \to (\mathsf{v},\mathsf{t}) \to \mathsf{Prop}\\
&\quad \textbf{where}\ \mathsf{WTNat} :: \forall \mathsf{n}.\ (\mathsf{NVal_F} \prec: \mathsf{v}, \mathsf{Functor}\ \mathsf{v},\\
&\qquad\qquad\qquad\qquad \mathsf{NTyp_F} \prec: \mathsf{t}, \mathsf{Functor}\ \mathsf{t})\\
&\qquad\qquad\qquad \Rightarrow \mathsf{WTNat_F}\ \mathsf{v}\ \mathsf{t}\ \mathsf{WTV}\ (\mathsf{vi}\ \mathsf{n}, \mathsf{tnat})
\end{aligned}
$$

This index is a pair (v, t) of a value and a type. As object-language values and types are themselves extensible, the corresponding meta-language types v and t are parameters of the WTNat functor.

To manipulate extensible logical relations, we need indexed algebras, fixpoints and operations:

```
type iAlg i (f :: (i → Prop) → (i → Prop)) a
  = ∀x :: i. f a x → a x
type iFix i (f :: (i → Prop) → (i → Prop)) (x :: i)
  = ∀a :: i → Prop. iAlg f a → a x ...
```

As these indexed variants are meant to construct logical relations, their parameters range over Prop instead of Set. Fortunately, this shift obviates the need for universal properties for iFix-ed values: it does not matter *how* a logical relation is built, but simply that it exists. Analogs to WF_Functor, WF_Algebra, and DistinctSubFunctor are similarly unnecessary.

Soundness of an Arithmetic Language

An example proof of type safety for the $\mathsf{Arith_F} \oplus \mathsf{Logic_F}$ language illustrates some of the implications of writing modular proofs with Church-encoded datatypes.

The previously defined eval function captures the operational semantics of this language in a modular way and reduces an expression to a $\mathsf{NVal_F} \oplus \mathsf{BVal_F} \oplus \mathsf{Stuck_F}$ value. Its type system is similarly captured by a modularly defined type-checking function typeof that *maybe* returns a $\mathsf{TNat_F} \oplus \mathsf{TBool_F}$ type representation:

```
data TNat_F t  = TNat
data TBool_F t = TBool
```

For this language soundness is formulated as:

Theorem soundness ::

$\forall$e t env. typeof e = Just t $\rightarrow$ WTValue (eval e env) t

The proof of this theorem is a fold of a proof algebra over the expression e which delegates the different cases to separate proof algebras for the different features. A summary of the most noteworthy aspects of these proofs follows.

Sublemmas The modular setting requires every case analysis to be captured in a sublemma. Because the superfunctor is abstract, the cases are not known locally and must be handled in a distributed fashion. Hence, modular lemmas built from proof algebras are not just an important tool for reuse in MTC – they are the main method of constructing extensible proofs.

Universal Properties Everywhere Universal properties are key to reasoning, and should thus be pervasively available throughout the framework. MTC has more infrastructure to support this.

As an example of their utility when constructing a proof, we may wish to prove a property of the extensible return value of an extensible function. Consider the $\mathsf{Logic_F}$ case of the soundness proof: given that typeof (If c e_1 e_2) = Some t_1, we wish to show that WTValue (eval (If c e_1 e_2)) t_1. If c evaluates to false, we need to show that WTValue e_2 t_1.

Since If c e_1 e_2 has type t_1, the definition of typeof says that e_1 has type t_1:

```
typeof_alg rec (If c e1 e2) =
  case project (rec c) of
    Just TBool →
      case (rec e1, rec e2) of
          (Just t1, Just t2) →
            if eq_type t1 t2 then Just t1 else Nothing
```

```
        _                     → Nothing
      Nothing → Nothing
```

In addition, the type equality test function, eq_{type}, says that e_1 and e_2 have the same type: $\mathsf{eq}_{type}\ \mathsf{t}_1\ \mathsf{t}_2 = \mathsf{true}$. We need to make use of a sublemma showing that $\forall \mathsf{t}_1\ \mathsf{t}_2.\ \mathsf{eq}_{type}\ \mathsf{t}_1\ \mathsf{t}_2 = \mathsf{true} \rightarrow \mathsf{t}_1 = \mathsf{t}_2$. As we have seen, in order to do so, the universal property must hold for typeof e_1. This is easily accomplished by packaging a proof of the universal property alongside t_1 in the typeof function.

Using universal properties is so important to reasoning that this packaging should be the default behavior, even though it is computationally irrelevant. Thankfully, packaging becomes trivial with the use of smarter constructors. These constructors have the additional advantage over standard smart constructors of being injective: $\mathsf{lit}\ \mathsf{j} = \mathsf{lit}\ \mathsf{k} \rightarrow \mathsf{j} = \mathsf{k}$, an important property for proving inversion lemmas. The proof of injectivity requires that the subterms of the functor have the universal property, established by the use of $\mathsf{in}'_{\mathsf{f}}$. To facilitate this packaging, we provide a type synonym that can be used in lieu of $\mathsf{Fix}_{\mathsf{M}}$ in function signatures:

```
type UP_F f = Functor f ⇒ Σ e.(UP f e)
```

Furthermore, the universal property should hold for any value subject to proof algebras, so it is convenient to include the property in all proof algebras. MTC provides a predicate transformer, UP_{P}, that captures this and augments induction principles accordingly.

```
UP_P :: Functor f ⇒
  (P :: ∀e. UP f e → Prop) → (e :: Fix_M f) → Σ e.(P e)
```

Equality and Universal Properties While packaging universal properties with terms enables reasoning, it does obfuscate equality of terms. In particular, two UP_{F} terms t and

t' may share the same underlying term (i.e., $\pi_1\ \mathsf{t} = \pi_1\ \mathsf{t}'$), while their universal property proof components are different.[3]

This issue shows up in the definition of the typing judgment for values. This judgment needs to range over $\mathsf{UP_F}\ \mathsf{f_v}$ values and $\mathsf{UP_F}\ \mathsf{f_t}$ types (where $\mathsf{f_v}$ and $\mathsf{f_t}$ are the value and type functors), because we need to exploit the injectivity of inject in our inversion lemmas. However, knowing $\mathsf{WTValue\ v\ t}$ and $\pi_1\ \mathsf{t} = \pi_1\ \mathsf{t}'$ no longer necessarily implies $\mathsf{WTValue\ v\ t}'$ because t and t' may have distinct proof components. To solve this, we make use of two auxiliary lemmas $WTV_{\pi_1,\mathsf{v}}$ and $WTV_{\pi_1,\mathsf{t}}$ that establish the implication[4]:

$$\begin{aligned}&\textbf{Theorem}\ WTV_{\pi_1,\mathsf{v}}\ (\mathsf{i} :: \mathsf{WTValue\ v\ t}) =\\ &\quad \forall \mathsf{v}'.\ \pi_1\ \mathsf{v} = \pi_1\ \mathsf{v}' \rightarrow \mathsf{WTValue\ v'\ t}\\ &\textbf{Theorem}\ WTV_{\pi_1,\mathsf{t}}\ (\mathsf{i} :: \mathsf{WTValue\ v\ t}) =\\ &\quad \forall \mathsf{t}'.\ \pi_1\ \mathsf{t}' = \pi_1\ \mathsf{t}' \rightarrow \mathsf{WTValue\ v\ t'}\end{aligned}$$

Similar lemmas are used for other logical relations. Features which introduce new rules need to also provide proofs showing that they respect this "safe projection" property.

4.4 Higher-Order Features

Binders and general recursion are ubiquitous in programming languages [Pie02], so it is important that our church encodings support these sorts of higher-order features. The untyped lambda calculus demonstrates the challenges of implementing both these features with extensible Church encodings.

4.4.1 Binders

Church encodings can support binders using a *parametric HOAS* (PHOAS) [Chl08] representation. PHOAS allows binders to be expressed as functors, while still preserving all the

[3]Actually, as proofs are opaque, we cannot tell if they are equal.
[4]Alternatively, we could assume proof irrelevance.

convenient properties of HOAS.

$\mathsf{Lambda_F}$ is a PHOAS-based functor for a feature with function application, abstraction and variables. The PHOAS style requires $\mathsf{Lambda_F}$ to be parameterized in the type v of variables, in addition to the usual type parameter r for recursive occurrences.

$$\textbf{data}\ \mathsf{Lambda_F\ v\ r = Var\ v \mid App\ r\ r \mid Lam\ (v \rightarrow r)}$$

As before, smart constructors build extensible expressions:

$$
\begin{aligned}
&\mathsf{var :: (Lambda_F\ v \prec: f) \Rightarrow v \rightarrow Fix_M\ f}\\
&\mathsf{var\ v = inject\ (Var\ v)}\\
&\mathsf{app :: (Lambda_F\ v \prec: f) \Rightarrow Fix_M\ f \rightarrow Fix_M\ f \rightarrow Fix_M\ f}\\
&\mathsf{app\ e_1\ e_2 = inject\ (App\ e_1\ e_2)}\\
&\mathsf{lam :: (Lambda_F\ v \prec: f) \Rightarrow (v \rightarrow Fix_M\ f) \rightarrow Fix_M\ f}\\
&\mathsf{lam\ f = inject\ (Lam\ f)}
\end{aligned}
$$

4.4.2 Non-Terminating Evaluation

Defining a semantics for the $\mathsf{Lambda_F}$ feature presents additional challenges. Evaluation of the untyped lambda-calculus can produce a closure, requiring a richer value type than before:

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Value} =\\
&\quad \mathsf{Stuck \mid I\ Nat \mid B\ Bool \mid Clos\ (Value \rightarrow Value)}
\end{aligned}
$$

Unfortunately, Coq does not allow such a definition, as the closure constructor is not strictly positive (recursive occurrences of Value occur both at positive and negative positions). Instead, a closure is represented as an expression to be evaluated in the context of an environment of variable-value bindings. The environment is a list of values indexed by variables represented as natural numbers Nat.

```
type Env v = [v]
```

The modular functor $\mathsf{Closure_F}$ integrates closure values into the framework of extensible values introduced in Section 4.1.5.

```
data Closure_F f a = Clos (Fix_M f) (Env a)
closure :: (Closure_F f ≺: r) ⇒
        Fix_M f → Env (Fix_M r) → Fix_M r
closure mf e = inject (Clos mf e)
```

To maintain consistency, only terminating functions can be defined in Coq. As the semantics of lambda is non-terminating, they can no longer be embedded in Coq as a function. One solution is to simply describe the semantics as a set of small-step or big-step reduction rules, which can be modularized using previously described techniques for indexed datatypes. Another alternative is to adapt the evaluation algebra to build functions that can be defined in Coq. The remainder of this section explores this alternative in order to illustrate how non-terminating functions can be adapted to work in our framework. A first attempt at defining an evaluation algebra is:

```
eval_Lambda ::  (Closure_F f ≺: r, Stuck_F ≺: r, Functor r) ⇒
    MAlgebra (Lambda_F Nat) (Env (Fix_M r) → Fix_M r)
eval_Lambda ⟦·⟧ exp env =
  case exp of
    Var index → env !! index
    Lam f     → closure (f (length env)) env
    App e1 e2 →
      case project $ ⟦e1⟧ env of
        Just (Clos e3 env′) → ⟦e3⟧ (⟦e2⟧ env : env′)
        _                   → stuck
```

The function $\mathsf{eval}_{\mathsf{Lambda}}$ instantiates the type variable v of the $\mathsf{Lambda}_{\mathsf{F}}\ \mathsf{v}$ functor with a natural number Nat, representing an index in the environment. The return type of the Mendler algebra is now a function that takes an environment as an argument. In the variable case there is an index that denotes the position of the variable in the environment, and $\mathsf{eval}_{\mathsf{Lambda}}$ simply looks up that index in the environment. In the lambda case $\mathsf{eval}_{\mathsf{Lambda}}$ builds a closure using f and the environment. Finally, in the application case, the expression e_1 is evaluated and analyzed. If that expression evaluates to a closure then the expression e_2 is evaluated and added to the closure's environment (env'), and the closure's expression e_3 is evaluated under this extended environment. Otherwise e_1 does not evaluate to a closure, and evaluation is stuck.

Unfortunately, this algebra is ill-typed on two accounts. Firstly, the lambda binder function f does not have the required type $\mathsf{Nat} \to \mathsf{Fix}_{\mathsf{M}}\ \mathsf{f}$. Instead, its type is $\mathsf{Nat} \to \mathsf{r}$, where r is universally quantified in the definition of the $\mathsf{MAlgebra}$ algebra. Secondly, and symmetrically, in the App case, the closure expression e_3 has type $\mathsf{Fix}_{\mathsf{M}}\ \mathsf{f}$ which does not conform to the type r expected by $[\![\cdot]\!]$ for the recursive call.

Both these symptoms have the same problem at their root. The Mendler algebra enforces inductive (structural) recursion by hiding that the type of the subterms is $\mathsf{Fix}_{\mathsf{M}}\ \mathsf{f}$ using universal quantification over r. Yet this information is absolutely essential for evaluating the binder: we need to give up structural recursion and use general recursion instead. This is unsurprising, as an untyped lambda term can be non-terminating.

Mixin algebras refine Mendler algebras with a more revealing type signature.

$$\textbf{type}\ \mathsf{Mixin\ t\ f\ a} = (\mathsf{t} \to \mathsf{a}) \to \mathsf{f\ t} \to \mathsf{a}$$

This algebra specifies the type t of subterms, typically $\mathsf{Fix}_{\mathsf{M}}\ \mathsf{f}$, the overall expression type. With this mixin algebra, $\mathsf{eval}_{\mathsf{Lambda}}$ is now well-typed:

$$\begin{aligned}&\mathsf{eval}_{\mathsf{Lambda}} :: (\mathsf{Closure}_{\mathsf{F}}\ \mathsf{e} \prec: \mathsf{v}, \mathsf{Stuck}_{\mathsf{F}} \prec: \mathsf{v}) \Rightarrow\\ &\quad \mathsf{Mixin}\ (\mathsf{Fix}_{\mathsf{M}}\ \mathsf{e})\ (\mathsf{Lambda}_{\mathsf{F}}\ \mathsf{Nat})\end{aligned}$$

```
    (Env (Fix_M v) → Fix_M v)
```

Mixin algebras have an analogous implementation to Eval as type classes, enabling all of MTC's previous composition techniques.

```
class Eval_X f g r where
  eval_xalg :: Mixin (Fix_M f) g (Env (Fix_M r) → Fix_M r)
instance (Stuck_F ≺: r, Closure_F f ≺: r, Functor r) ⇒
          Eval_X f (Lambda_F Nat) r where
  eval_xalg = eval_Lambda
```

Although the code of $\mathsf{eval}_{\mathsf{Lambda}}$ still appears generally recursive, it is actually *not* because the recursive calls are abstracted as a parameter (like with Mendler algebras). Accordingly, $\mathsf{eval}_{\mathsf{Lambda}}$ does not raise any issues with Coq's termination checker. Mixin algebras resemble the *open recursion* style which is used to model inheritance and mixins in object-oriented languages [Coo89]. Still, Mendler encodings only accept Mendler algebras, so using mixin algebras with Mendler-style encodings requires a new form of fold.

In order to overcome the problem of general recursion, the open recursion of the mixin algebra is replaced with a bounded inductive fixpoint combinator, boundedFix, that returns a default value if the evaluation does not terminate after n recursion steps.

```
boundedFix :: ∀f a.Functor f ⇒ Nat → a →
                   Mixin (Fix_M f) f a → Fix_M f → a
boundedFix n def alg e =
  case n of
    0  → def
    m → alg (boundedFix (m − 1) def alg) (out_f e)
```

The argument e is a Mendler-encoded expression of type Fix_M f. boundedFix first uses out_f to unfold the expression into a value of type f (Fix_M f) and then applies the algebra to that value recursively. In essence boundedFix can define *generally* recursive operations by case analysis, since it can inspect values of the recursive occurrences. The use of the bound prevents non-termination.

Bounded Evaluation Evaluation can now be modularly defined as a bounded fixpoint of the mixin algebra Eval_X. The definition uses a distinguished bottom value, $\bot$, that represents a computation which does not finish within the given bound.

```
data ⊥_F a = Bot
⊥= inject Bot
eval_X :: (Functor f, ⊥_F ≺: r, Eval_X f f r) ⇒
   Nat → Fix_M f → Env → Fix_M r
eval_X n e env = boundedFix n (λ_ →⊥) eval_xalg e env
```

4.4.3 Backwards compatibility

The higher-order PHOAS feature has introduced a twofold change to the algebras used by the evaluation function:

1. eval_X uses mixin algebras instead of Mendler algebras.
2. eval_X now expects algebras over a parameterized functor.

The first change is easily accommodated because Mendler algebras are compatible with mixin algebras. If a non-binder feature defines evaluation in terms of a Mendler algebra, it does not have to define a second mixin algebra to be used alongside binder features. The mendlerToMixin function automatically derives the required mixin algebra from the Mendler algebra.

```
mendlerToMixin :: MAlgebra f a → Mixin (Fix_M g) f a
mendlerToMixin alg = alg
```

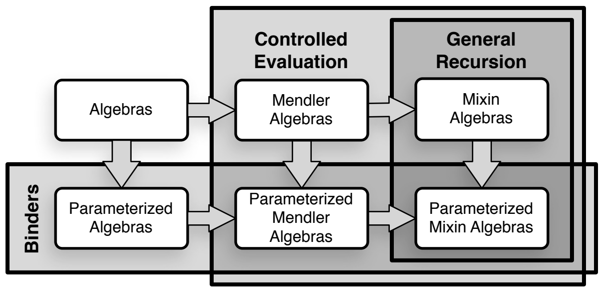


Figure 4.8: Hierarchy of Algebra Adaptation

This conversion function can be used to adapt evaluation for the arithmetic feature to a mixin algebra:

$\textbf{instance}\ \mathsf{Eval\ Arith_F\ f} \Rightarrow \mathsf{Eval_X\ f\ Arith_F\ r}\ \textbf{where}$
  $\mathsf{eval_{xalg}}\ [\![\cdot]\!]\ \mathsf{e\ env} =$
    $\mathsf{mendlerToMixin\ evalAlgebra\ (flip}\ [\![\cdot]\!]\ \mathsf{env)\ e}$

The algebras of binder-free features can be similarly adapted to build an algebra over a parametric functor. Figure 4.8 summarizes the hierarchy of algebra adaptations. Non-parameterized Mendler algebras are the most flexible because they can be adapted and reused with both mixin algebras and parametric superfunctors. They should be used by default, only resorting to mixin algebras when necessary.

4.5 Reasoning with Higher-Order Features

The switch to a bounded evaluation function over parameterized Church encodings requires a new statement of soundness.

Theorem $\mathsf{soundness_X} :: \forall \mathsf{f}\ \mathsf{f_t}\ \mathsf{env}\ \mathsf{t}\ \Gamma\ \mathsf{n}.$
$\forall \mathsf{e_1} :: \mathsf{Fix_M\ (f\ (Maybe\ (Fix_M\ f_t)))}.$
$\forall \mathsf{e_2} :: \mathsf{Fix_M\ (f\ Nat)}.$

$\Gamma \vdash \mathsf{e}_1 \equiv \mathsf{e}_2 \rightarrow \mathsf{WF\_Environment}\ \Gamma\ \mathsf{env} \rightarrow$

$\mathsf{typeof}\ \mathsf{e}_1 = \mathsf{Just}\ \mathsf{t} \rightarrow \mathsf{WTValue}\ (\mathsf{eval}_\mathsf{X}\ \mathsf{n}\ \mathsf{e}_2\ \mathsf{env})\ \mathsf{t}$

The proof of $\mathsf{soundness}_\mathsf{X}$ features two substantial changes to the proof of $\mathsf{soundness}$ from Section 4.3.2.

4.5.1 Proofs over Parametric Church Encodings

The statement of $\mathsf{soundness}_\mathsf{X}$ uses two instances of the same PHOAS expression $\mathsf{e}{::}\forall \mathsf{v}.\mathsf{Fix}_\mathsf{M}\ (\mathsf{f}\ \mathsf{v})$. The first, e_1, instantiates v with the appropriate type for the typing algebra, while e_2 instantiates v for the evaluation algebra.

In recursive applications of $\mathsf{soundness}_\mathsf{X}$, the connection between e_1 and e_2 is no longer apparent. As they have different types, Coq considers them to be distinct, so case analysis on one does not convey information about the other. Chlipala [Chl08] shows how the connection can be retained with the help of an auxiliary equivalence relation $\Gamma \vdash \mathsf{e}_1 \equiv \mathsf{e}_2$, which uses the environment Γ to keep track of the current variable bindings. The top-level application, where the common origin of e_1 and e_2 is apparent, can easily supply a proof of this relation. By induction on this proof, recursive applications of $\mathsf{soundness}_\mathsf{X}$ can then analyze e_1 and e_2 in lockstep. Figure 4.9 shows the rules for determining equivalence of lambda expressions.

$$\frac{(x,x') \in \Gamma}{\Gamma \vdash \mathsf{var}\ x \equiv \mathsf{var}\ x'}\ (\textsc{Eqv-Var}) \qquad \frac{\Gamma \vdash e_1 \equiv e_1' \qquad \Gamma \vdash e_2 \equiv e_2'}{\Gamma \vdash \mathsf{app}\ e_1\ e_2 \equiv \mathsf{app}\ e_1'\ e_2'}\ (\textsc{Eqv-App})$$

$$\frac{\forall x x'.(x,x'),\Gamma \vdash f(x) \equiv f'(x')}{\Gamma \vdash \mathsf{lam}\ f \equiv \mathsf{lam}\ f'}\ (\textsc{Eqv-Abs})$$

Figure 4.9: Lambda Equivalence Rules

4.5.2 Proofs for Non-Inductive Semantic Functions

Proofs for semantic functions that use $\mathsf{boundedFix}$ proceed by induction on the bound. Hence, the reasoning principle for mixin-based bounded functions f is in general: provided a base case $\forall e, P(f\ 0\ e)$, and inductive case $\forall n\ e, (\forall e', P(f\ n\ e')) \rightarrow \forall e, P(f\ (n+1)\ e)$ hold, $\forall n\ e, P(f\ n\ e)$ also holds.

In the base case of $\mathsf{soundness}_\mathsf{X}$, the bound has been reached and eval_X returns $\bot$. The proof of this case relies on adding to the $\mathsf{WTValue}$ judgment the WF-Bot rule stating that every type is inhabited by $\bot$.

$$\frac{}{\vdash \bot : T} \qquad \text{(WF-Bot)}$$

Hence, whenever evaluation returns $\bot$, soundness trivially holds.

The inductive case is handled by a proof algebra whose statement includes the inductive hypothesis provided by the induction on the bound: $\mathsf{IH} :: \forall n\ e, (\forall e', P(f\ n\ e')) \rightarrow P(f\ (n+1)\ e)$. The $\mathsf{App}\ \mathsf{e}_1\ \mathsf{e}_2$ case of the soundness theorem illustrates the reason for including IH in the statement of the proof algebra. After using the induction hypothesis to show that $\mathsf{eval}_\mathsf{X}\ \mathsf{e}_1\ \mathsf{env}$ produces a well-formed closure $\mathsf{Clos}\ \mathsf{e}_3\ \mathsf{env}'$, we must then show that evaluating e_3 under the $(\mathsf{eval}_\mathsf{X}\ \mathsf{e}_2\ \mathsf{env}) : \mathsf{env}'$ environment is also well-formed. However, e_3 is not a subterm of $\mathsf{App}\ \mathsf{e}_1\ \mathsf{e}_2$, so the conventional induction hypothesis for subterms does not apply. Because $\mathsf{eval}_\mathsf{X}\ \mathsf{e}_3\ ((\mathsf{eval}_\mathsf{X}\ \mathsf{e}_2\ \mathsf{env}) : \mathsf{env}')$ is run with a smaller bound, the bounded induction hypothesis IH can be used.

4.5.3 Proliferation of Proof Algebras

In order to incorporate non-parametric inductive features in the $\mathsf{soundness}_\mathsf{X}$ proof, existing proof algebras for those features need to be adapted. To cater to the four possible proof signatures of soundness (one for each definition of $[\![\cdot]\!]$), a naive approach requires four different proof algebras for an inductive non-parametric feature.[5] This is not acceptable,

[5]Introducing type-level binders would further compound the situation with four possible signatures for the typeof algebra.

because reasoning about a feature's soundness should be independent of how a language adapts its evaluation algebra. Hence, MTC allows features to define a single proof algebra, and provides the means to adapt and reuse that proof algebra for the four signature variants. These proof algebra adaptations rely on *mediating type class instances* which automatically build an instance of the new proof algebra from the original proof algebra.

Adapting Proofs to Parametric Functors

Adapting a proof algebra over the expression functor to one over the indexed functor for the equivalence relation first requires a definition of equivalence for non-parametric functors. Fortunately, equivalence for any such functor $\mathsf{f_{np}}$ can be defined generically:

$$\frac{\Gamma \vdash \overline{a} \equiv \overline{b}}{\Gamma \vdash \mathsf{inject}(\mathsf{C}\ \overline{a}) \equiv \mathsf{inject}(\mathsf{C}\ \overline{b})} \qquad (\textsc{Eqv-NP})$$

Eqv-NP states that the same constructor C of $\mathsf{f_{np}}$, applied to equivalent subterms $\bar{a}$ and $\bar{b}$, produces equivalent expressions.

The mediating type class adapts $\mathsf{f_{np}}$ proofs of propositions on two instances of the same PHOAS expression, like soundness, to proof algebras over the parametric functor.

$\textbf{instance}\ (\mathsf{PAlg\ N\ P\ f_{np}}) \Rightarrow \mathsf{iPAlg\ N\ P}\ (\textsc{Eqv-NP}\ \mathsf{f_{np}})$

This instance requires a small concession: proofs over $\mathsf{f_{np}}$ have to be stated in terms of two expressions with distinct superfunctors f and $\mathsf{f'}$ rather than two occurrences of the same expression. Induction over these two expressions requires a variant of PAlg for pairs of fixpoints.

Adapting Proofs to Non-Inductive Semantic Functions

To be usable regardless of whether $\mathsf{fold_M}$ or $\mathsf{boundedFix}$ is used to build the evaluation function, an inductive feature's proof needs to reason over an abstract fixpoint operator and induction principle. This is achieved by only considering a single step of the evaluation algebra and leaving the recursive call abstract:

$\textbf{type}$ soundness e tp ev =
  $\forall$env t.tp (out$_{\mathsf{f}}$ (π_1 e)) = Just t $\rightarrow$
    WTValue (ev (out_t$'$ (π_1 e)) env) t)

$\textbf{type}$ soundness$_{alg}$ rec$_t$ rec$_e$
  (typeof$_{alg}$ :: Mixin (Fix$_{\mathsf{M}}$ f) f (Maybe (Fix$_{\mathsf{M}}$ t)))
  (eval$_{alg}$ :: Mixin (Fix$_{\mathsf{M}}$ f) f (Env (Fix$_{\mathsf{M}}$ r) $\rightarrow$ Fix$_{\mathsf{M}}$ r))
  (e :: Fix$_{\mathsf{M}}$ f) (e_UP$'$ :: UP e) =
    $\forall$IHc :: ($\forall$e$'$.
      soundness e$'$ (typeof$_{alg}$ rec$_t$) (eval$_{alg}$ rec$_e$) $\rightarrow$
      soundness e$'$ rec$_t$ rec$_e$).
    soundness e (typeof$_{alg}$ rec$_t$) (eval$_{alg}$ rec$_e$)

The hypothesis IHc is used to relate calls of rec$_e$ and rec$_t$ to applications of eval$_{alg}$ and typeof$_{alg}$.

A mediating type class instance again lifts a proof algebra with this signature to one that includes the Induction Hypothesis generated by induction on the bound of boundedFix.

$\textbf{instance}$ (PAlg N P E) $\Rightarrow$ iPAlg N (IH $\rightarrow$ P) E

4.6 Case Studies

As a demonstration of the utility of the extensible datatypes provided by MTC, we have built a set of five reusable language features and combined them into a mini-ML [CDKD86] variant. The study also builds five other languages from these features.[6] Figure 5.12 presents the syntax of the expressions, values, and types provided by the features; each line is annotated with the feature that provides that set of definitions.

The Coq files that implement these features average roughly 1100 LoC and come with a typing and evaluation function in addition to soundness and continuity proofs. Each

[6]Also available at `http://www.cs.utexas.edu/~bendy/MTC`

language needs on average only 100 LoC to build its semantic functions and soundness proofs from the files implementing its features. The framework itself consists of about 2500 LoC.

$$\begin{array}{lll}
\texttt{e} ::= & \mathbb{N} \mid \texttt{e + e} & \textit{Arith} \\
\mid & \mathbb{B} \mid \textbf{if } \texttt{e} \textbf{ then } \texttt{e} \textbf{ else } \texttt{e} & \textit{Bool} \\
\mid & \textbf{case } \texttt{e} \textbf{ of } \{ \ \mathbf{z} \Rightarrow \texttt{e} \ ; \ \mathbf{S} \ \texttt{n} \Rightarrow \texttt{e}\} & \textit{NatCase} \\
\mid & \textbf{lam } \texttt{x : T.e} \mid \texttt{e e} \mid \texttt{x} & \textit{Lambda} \\
\mid & \textbf{fix } \texttt{x : T.e} & \textit{Recursion}
\end{array}$$

$$\begin{array}{lll}
\texttt{V} ::= & \mathbb{N} & \textit{Arith} \\
\mid & \mathbb{B} & \textit{Bool} \\
\mid & \textbf{closure } \texttt{e} \ \overline{\texttt{V}} & \textit{Lambda}
\end{array}
\qquad
\begin{array}{lll}
\texttt{T} ::= & \textbf{nat} & \textit{Arith} \\
\mid & \texttt{bool} & \textit{Bool} \\
\mid & \texttt{T} \rightarrow \texttt{T} & \textit{Lambda}
\end{array}$$

Figure 4.10: mini-ML expressions, values, and types

The generic soundness proof, reused by each language, relies on a proof algebra to handle the case analysis of the main lemma. Each case is handled by a sub-algebra. These sub-algebras have their own set of proof algebras for case analysis or induction over an abstract superfunctor. The whole set of dependencies of a top-level proof algebra forms a *proof interface* that must be satisfied by any language which uses that algebra.

Such proof interfaces introduce the problem of *feature interactions* [BKH11], well-known from modular component-based frameworks. In essence, a feature interaction is functionality (e.g., a function or a proof) that is only necessary when two features are combined. An example from this study is the inversion lemma which states that values with type **nat** are natural numbers: $\vdash \texttt{x} : \textbf{nat} \rightarrow \texttt{x} :: \mathbb{N}$. The Bool feature introduces a new typing judgment, WT-Bool for boolean values. Any language which includes both these features must have an instance of this inversion algebra for WT-Bool. Our modular approach supports feature interactions by capturing them in type classes. A missing case, like for WT-Bool, can then be easily added as a new instance of that type class, without affecting or overriding existing code.

In this case study, feature interactions consist almost exclusively of inversion principles for judgments and the projection principles of Section 4.3.2. Thankfully, their proofs are relatively straightforward and can be dispatched by tactics hooked into the type class inference algorithm. These tactics help minimize the number of interaction type class instances, which could otherwise easily grow exponentially in the number of features.

Chapter 5

Effect Modularity in Mechanized Metatheory

The previous chapter demonstrated how Church-encoded datatypes abstracted over a superfunctor could be easily extended with new constructors. This allows features to be modularized along the dimension of *values* by enabling the syntactic and semantic domains of a programming language to be expressed as the union of the syntactic and semantic values of distinct features. While this often suffices to modularize syntactic domains, changes to the syntactic values of a programming language can induce changes in the dimension of *effects* in the semantic domain.

As an example, consider adding the syntax and reduction rules for arithmetic expressions and references:

$$\begin{array}{l|l}
\begin{array}{l}
\mathsf{e} ::= \mathbb{N} \mid \mathtt{e\ +\ e} \\
\mathsf{v} ::= \mathbb{N}
\end{array}
&
\begin{array}{l}
\mathsf{e} ::= \mathtt{ref\ e} \mid \mathtt{!e} \mid \mathtt{e\ :=\ e} \\
\mathsf{v} ::= \mathbb{L} \mid \mathtt{unit}
\end{array}
\\[1em]
\dfrac{}{\mathsf{n} \Downarrow \mathsf{n}}
&
\dfrac{\mathsf{e} \mid \sigma \ \Downarrow\ \mathsf{v} \mid \sigma' \qquad l \notin \sigma'}{\mathsf{ref\ e} \mid \sigma \ \Downarrow\ \mathsf{l} \mid (\sigma', \mathsf{l} \mapsto \mathsf{v})}
\quad
\dfrac{\mathsf{e} \mid \sigma \ \Downarrow\ \mathsf{l} \mid \sigma' \qquad \sigma'(l) = v}{\mathsf{!e} \mid \sigma \ \Downarrow\ \mathsf{v} \mid \sigma'}
\\[1em]
\dfrac{\mathsf{e_m} \ \Downarrow\ \mathsf{m} \qquad \mathsf{e_n} \ \Downarrow\ \mathsf{n}}{\mathsf{e_m} + \mathsf{e_n} \ \Downarrow\ \mathsf{m} + \mathsf{n}}
&
\dfrac{\mathsf{e_1} \mid \sigma \ \Downarrow\ \mathsf{l} \mid \sigma_1 \qquad \mathsf{e_2} \mid \sigma_1 \ \Downarrow\ \mathsf{v} \mid \sigma_2}{\mathsf{e_1} := \mathsf{e_2} \mid \sigma \ \Downarrow\ \mathsf{unit} \mid [\mathsf{l} \mapsto \mathsf{v}]\sigma_2}
\end{array}$$

Both features extend the syntactic domain e and the set of values v, but the semantic domain of the reduction relation for references includes additional information about a variable store σ: $\Downarrow\ ::\ \mathsf{e} \rightarrow \sigma \rightarrow \mathsf{v} \times \sigma$. In order to be compatible with references, the reduction rules for arithmetic expressions must also account for the variable store. Additional features, such as exceptions, can impose additional constraints on the effects of the semantic domain. The semantic domains of features which require different sets of semantic effects have to be extended in order for the features to be compatible. If the semantic domain of the features each have a *fixed* set of effects, this entails manually modifying the domain (and by extension, any meta-theory proofs involving the semantic domain). One alternative which preserves feature modularity is to *abstract* semantic domains over values *and* effects, constraining the set of effects to include those required by the feature.

The functional programming community has embraced representing effects using monads. Moggi showed how many computational effects could be expressed as monads [Mog91]. Wadler helped popularize the approach as a means of extending interpreters [Wad92a]. Abstracting a semantic domain over a monad allows that domain to be extended to support additional effects. This chapter demonstrates how to modularly reason about these abstract monadic semantic domains, enabling feature modularity for mechanized meta-theory of languages with effects.

5.1 The M^{3}TL Monad Library

The *Modular Monadic MetaTheory Library*(M^{3}TL) integrates monads into the MTC framework in order to support semantic domains with extensible effects using *monads* and *monad transformers*. This library is inspired by the Haskell *monad transformer library* (MTL) [LHJ95]. Monads provide a uniform representation for encapsulating computational effects such as mutable state, exception handling, and non-determinism. Monad transformers allow monads supporting the desired set of effects to be built. M^{3}TL also provides *algebraic laws* for reasoning about monadic definitions. These algebraic laws, which can

Monad class

```
class Functor m ⇒ Monad m where
  return      :: a → m a
  (≫=)        :: m a → (a → m b) → m b
  return_bind :: return x ≫= f ≡ f x
  bind_return :: p ≫= return ≡ p
  bind_bind   :: (p ≫= f) ≫= g ≡
                   p ≫= λx → (f x ≫= g)
  fmap_bind   :: fmap f t ≡ t ≫= (return ∘ f)
```

Failure class

```
class Monad m ⇒ FailMonad m where
  fail      :: m a
  bind_fail :: fail ≫= f ≡ fail
```

State class

```
class Monad m ⇒ 𝕊_M s m where
  get       :: m s
  put       :: s → m ()
  get_query :: get ≫ t ≡ t
  put_get   :: put s ≫ get ≡ put s ≫ return s
  get_put   :: get ≫= put ≡ return ()
  get_get   :: get ≫= λs ≫= get ≫= f s ≡
                 get ≫= λs → f s s
  put_put   :: put s1 ≫ put s2 ≡ put s2
```

Identity monad

```
newtype 𝕀 a

𝕀    :: a → 𝕀 a
run𝕀 :: 𝕀 a → a
```

Failure transformer

```
newtype FailT m a

FailT    :: m (Maybe a) → FailT m a
runFailT :: FailT m a → m (Maybe a)
```

State transformer

```
newtype 𝕊_T s m a

𝕊_T    :: (s → m (a, s)) → 𝕊_T s m a
run𝕊_T :: 𝕊_T s m a → s → m (a, s)
```

Reader class

```
class Monad m ⇒ ℝ_M e m where
  ask          :: m e
  local        :: (e → e) → m a → m a
  ask_query    :: ask ≫ t ≡ t
  local_return :: local f ∘ return = return
  ask_ask      :: ask ≫= λs ≫= ask ≫= f s ≡
                    ask ≫= λs → f s s
  ask_bind     :: t ≫= λx → ask ≫= λe → f x e ≡
                    ask ≫= λe → t ≫= λx → f x e
  local_bind   :: local f (t ≫= g) ≡
                    local f t ≫= local f ∘ g
  local_ask    :: local f ask ≡ ask ≫= return ∘ f
  local_local  :: local f ∘ local g ≡ local (g ∘ f)
```

Exception class

```
class Monad m ⇒ 𝔼_M x m where
  throw          :: x → m a
  catch          :: m a → (x → m a) → m a
  bind_throw     :: throw e ≫= f ≡ throw e
  catch_throw_1  :: catch (throw e) h ≡ h e
  catch_throw_2  :: catch t throw ≡ t
  catch_return   :: catch (return x) h ≡ return x
  fmap_catch     :: fmap f (catch t h) ≡
                      catch (fmap f t) (fmap f ∘ h)
```

Exception transformer

```
newtype 𝔼_T x m a

𝔼_T    :: m (Either x a) → 𝔼_T x m a
run𝔼_T :: 𝔼_T x m a → m (Either x a)
```

Reader transformer

```
newtype ℝ_T e m a

ℝ_T :: (e → m a) → ℝ_T e m a
runℝ_T :: ℝ_T e m a → e → m a
```

Figure 5.1: Key classes, definitions and laws from M^{3}TL. The names of algebraic laws are in bold.

only be documented informally in the MTL, are fully integrated into M^{3}TL's type classes using Coq's expressive dependent types. The library systematically includes laws for all monad subclasses, several of which have not been covered in the functional programming literature before.

Library overview Figure 5.1 summarizes the library's key classes, definitions and laws. The type class Monad describes the basic interface of monads. The type m a denotes computations of type m which produce values of type a when executed. The function return lifts a value of type a into a (pure) computation that simply produces the value. The *bind* function $\gg\!=$ composes a computation m a producing values of type a, with a function that accepts a value of type a and returns a computation of type m b. The function $\gg$ defines a special case of bind that discards the intermediate value:

$$(\gg) :: \mathsf{Monad\ m} \Rightarrow \mathsf{m\ a} \to \mathsf{m\ b} \to \mathsf{m\ b}$$
$$\mathsf{ma} \gg \mathsf{mb} = \mathsf{ma} \gg\!= \lambda\_ \to \mathsf{mb}$$

The **do** notation is syntactic sugar for the *bind* operator: **do** $\{\mathsf{x} \leftarrow \mathsf{f}; \mathsf{g}\}$ means $\mathsf{f} \gg\!= \lambda \mathsf{x} \to \mathsf{g}$. This notation is included in M^{3}TL using Coq's Notation mechanism.

Particular monads can be built from basic monad types such as the identity monad ($\mathbb{I}$) and monad transformers including the failure (FailT), mutable state ($\mathbb{S}_{\mathrm{T}}$), and exception ($\mathbb{E}_{\mathrm{T}}$) transformers. These transformers are combined into different monad stacks with $\mathbb{I}$ at the bottom. Constructor and extractor functions such as $\mathbb{S}_{\mathrm{T}}$ and $run\mathbb{S}_{\mathrm{T}}$ provide the signatures of the functions for building and running particular monads/transformers.

In order to support extensible effects, the semantic domain needs to be abstracted over the monad implementation used. Any implementation which includes the required operations is valid. These operations are captured in type classes such as $\mathbb{S}_{\mathrm{M}}$ and $\mathbb{E}_{\mathrm{M}}$, also called *monad subclasses.* The type classes (denoted by subscript M) require a monad stack to support a particular effect without assuming a particular stack configuration.[1] Each

[1]Supporting two instances of the same effect requires extra machinery [Hue97].

class offers a set of primitive operations, such as get to access the state for $\mathbb{S}_{\mathrm{M}}$.

Algebraic laws Each monad (sub)class includes a set of algebraic laws that govern its operations. These laws are an integral part of the definition of the monad type classes and constrain the possible implementations to sensible ones. Thus, even without knowing the particular implementation of the abstract monad, we can still modularly reason about its behavior via these laws. This is crucial for supporting modular reasoning [OSC10].

The first three laws for the Monad class are standard, while the last law (fmap_bind) relates fmap and bind in the usual way. Each monad subclass also includes its own set of laws. The laws for various subclasses can be found scattered throughout the functional programming literature, such as for failure [GH11] and state [OSC10, GH11], but in order to support reasoning over the abstract monad, M^{3}TL is forced to systematically bring them together. Although most of M^{3}TL's laws have been presented in the semantics literature in one form or another, some of the laws have not appeared in the functional programming literature. One such example are the laws for the exception class:

- The bind_throw law generalizes the bind_fail law: a sequential computation is aborted by throwing an exception.
- The catch_throw$_1$ law states that the exception handler is invoked when an exception is thrown in a catch.
- The catch_throw$_2$ law indicates that an exception handler is redundant if it just re-throws the exception.
- The catch_return law states that a catch around a pure computation is redundant.
- The fmap_catch law states that pure functions (fmap f) distribute on the right with catch.

– Simplified value interface –

```
type Value
loc   :: Int → Value
stuck :: Value
unit  :: Value
isLoc :: Value → Maybe Int
```

—— Expression functor ——

```
data RefF a = Ref a
    | DeRef a
    | Assign a a
```

— Monadic typing algebra —

```
typeofRef :: FailMonad m ⇒
  AlgebraM RefF (m Type)
typeofRef rec (Ref e) =
  do t ← rec e
     return (tRef t)
typeofRef rec (DeRef e) =
  do te ← rec e
     maybe fail return (isTRef te)
typeofRef rec (Assign e1 e2) =
  do t1 ← rec e1
     case isTRef t1 of
       Nothing → fail
       Just t → do t2 ← rec e2
                   if (t ≡ t2)
                     then return tUnit
                     else fail
```

——— Simplified type interface ———

```
type Type
tRef   :: Type → Type
tUnit  :: Type
isTRef :: Type → Maybe Type


type Store = [Value]
```

—— Monadic evaluation algebra ——

```
evalRef :: SM Store m ⇒ AlgebraM RefF (m Value)
evalRef rec (Ref e) =
  do v   ← rec e
     env ← get
     put (v : env)
     return (loc (length env))
evalRef rec (DeRef e) =
  do v   ← rec e
     env ← get
     case isLoc v of
       Nothing → return stuck
       Just n → return (maybe stuck id (fetch n env))
evalRef rec (Assign e1 e2) =
  do v   ← rec e1
     env ← get
     case isLoc v of
       Nothing → return stuck
       Just n   → do v2 ← rec e2
                     put (replace n v2 env)
                     return unit
```

Figure 5.2: Syntax, type, and semantic function definitions for references.

5.2 Modular Monadic Semantics

Features can utilize M^{3}TL to construct semantic domains which are extensible with both new values and new effects. A modular monadic evaluation algebra exposes these two dimensions:

$$\mathsf{evalRef} :: (\mathsf{Unit} \prec: \mathsf{v}, \mathbb{S}_{\mathrm{M}}\ \mathsf{Store}\ \mathsf{m}) \Rightarrow \mathsf{Algebra_M}\ \mathsf{RefF}\ (\mathsf{m}\ \mathsf{v})$$
$$\mathsf{evalErr} :: \mathbb{E}_{\mathrm{M}}\ ()\ \mathsf{m} \Rightarrow \mathsf{Algebra_M}\ \mathsf{ErrF}\ (\mathsf{m}\ \mathsf{v})$$

These algebras use monad subclasses such as $\mathbb{S}_{\mathrm{M}}$ and $\mathbb{E}_{\mathrm{M}}$ to *constrain* the monad required by the feature, allowing the monad to have more effects than those used in the feature.

The set of values v is similarly constrained to include Unit by the evaluation algebra for references. These two algebras can be combined to create a new evaluation algebra whose domain supports both types of effects:

$$(\mathsf{Unit} \prec: \mathsf{v}, \mathbb{S}_{\mathrm{M}}\ \mathsf{m\ s}, \mathbb{E}_{\mathrm{M}}\ \mathsf{m\ x}) \Rightarrow \mathsf{MAlgebra}\ (\mathsf{RefF} \oplus \mathsf{ErrF})\ (\mathsf{m\ v})$$

The combination imposes both type class constraints and the constraint on values while the monad type and type of values remains extensible with new effects and values. Figure 5.3 gives the complete set of effects used by the evaluation functions for the five language features used in the case study of Section 5.5.

Arithmetic Expressions	Monad m
Boolean Expressions	Monad m
Errors	$\mathbb{E}_{\mathrm{M}}$ () m
References	$\mathbb{S}_{\mathrm{M}}$ Store m
Lambda	$\mathbb{R}_{\mathrm{M}}$ Env m, FailMonad m

Figure 5.3: Effects used by the case study's evaluation algebras.

Figure 5.2 illustrates this approach with definitions for the functor for expressions and the evaluation and typing algebras for the reference feature. Other features have similar definitions.

The type RefF is the functor for references. It has constructors for creating references (Ref), dereferencing (DeRef) and assigning (Assign) references. The evaluation algebra evalRef uses the state monad for its reference environment, which is captured in the type class constraint $\mathbb{S}_{\mathrm{M}}$ Store m. The typing algebra (typeofRef) is also monadic, using the failure monad to denote ill-typing.

5.2.1 Effect-Dependent Theorems

Monadic semantic domains are compatible with new effects, and algebraic laws facilitate writing extensible proofs over these monadic algebras. Effects introduce further challenges to proof reuse, however: each combination of effects induces its own type soundness statement. Consider the theorem for a language with references which features a store σ and a store typing Σ that are related through the store typing judgement $\Sigma \vdash \sigma$:

$$\boxed{\begin{array}{c} \forall \mathsf{e}, \mathsf{t}, \Sigma, \sigma. \left\{ \begin{array}{c} \mathsf{typeof\ e} \equiv \mathsf{return\ t} \\ \Sigma \vdash \sigma \end{array} \right\} \rightarrow \\ \exists \mathsf{v}, \Sigma', \sigma'. \left\{ \begin{array}{c} \mathsf{put}\ \sigma \gg [\![\mathsf{e}]\!] \equiv \mathsf{put}\ \sigma' \gg \mathsf{return\ v} \\ \Sigma' \supseteq \Sigma \quad \wedge \quad \Sigma' \vdash \mathsf{v} : \mathsf{t} \quad \wedge \quad \Sigma' \vdash \sigma' \end{array} \right\} \qquad (\text{LSound}_S) \end{array}}$$

Contrast this with the theorem for a language with errors, which must account for the computation possibly ending in an exception being thrown:

$$\boxed{\begin{array}{c} \forall \mathsf{e}, \mathsf{t}.\ \mathsf{typeof\ e} \equiv \mathsf{return\ t} \rightarrow \\ (\exists \mathsf{v}.\ [\![\mathsf{e}]\!] \equiv \mathsf{return\ v} \wedge \vdash \mathsf{v} : \mathsf{t}) \vee (\exists \mathsf{x}.\ [\![\mathsf{e}]\!] \equiv \mathsf{throw\ x}) \qquad (\text{LSound}_E) \end{array}}$$

Clearly, the available effects are essential for the formulation of the theorem. A larger language which involves both exceptions and state requires yet another theorem where the impacts of both effects cross-cut one another[2]:

$$\boxed{\begin{array}{c} \forall \mathsf{e}, \mathsf{t}, \Sigma, \sigma. \left\{ \begin{array}{c} \mathsf{typeof\ e} \equiv \mathsf{return\ t} \\ \Sigma \vdash \sigma \end{array} \right\} \rightarrow \\ \exists \mathsf{v}, \Sigma', \sigma'. \left\{ \begin{array}{c} \mathsf{put}\ \sigma \gg [\![\mathsf{e}]\!] \equiv \mathsf{put}\ \sigma' \gg \mathsf{return\ v} \\ \Sigma' \supseteq \Sigma \quad \wedge \quad \Sigma' \vdash \mathsf{v} : \mathsf{t} \quad \wedge \quad \Sigma' \vdash \sigma' \end{array} \right\} \\ \vee \quad \exists \mathsf{x}.\ \mathsf{put}\ \sigma \gg [\![\mathsf{e}]\!] \equiv \mathsf{throw\ x} \qquad (\text{LSound}_{ES}) \end{array}}$$

Modular formulations of LSound$_E$ and LSound$_S$ are useless for proving a modular variant of LSound$_{ES}$ because their induction hypotheses have the wrong form. The hypoth-

[2]A similar proliferation of soundness theorems can be found in TAPL [Pie02].

$$\frac{}{\Sigma \vdash_{\mathsf{M}} \mathsf{v_m} \ : \ \mathsf{fail}} \quad (\text{WFM-Illtyped})$$

$$\frac{\Sigma \vdash \mathsf{v} \ : \ \mathsf{t}}{\Sigma \vdash_{\mathsf{M}} \mathsf{return\ v} \ : \ \mathsf{return\ t}} \quad (\text{WFM-Return})$$

Figure 5.4: Typing rules for pure monadic values.

esis for LSound$_E$ requires the result to be of the form return v, disallowing put $\sigma' \gg$ return v (the form required by LSound$_S$). Similarly, the hypothesis for LSound$_S$ does not account for exceptions occurring in subterms. In general, without anticipating additional effects, type soundness theorems with fixed sets of effects cannot be reused modularly.

5.3 Modular Monadic Type Soundness

In order to preserve a measure of modularity, we do not prove type soundness directly for a given feature, but by means of a more generic theorem. The technique of proving a theorem of interest by means of a more general theorem is well-known. For a conventional monolithic language, for instance, type soundness is often established for any well-formed typing context, even though the main interest lies with the more specific initial, empty context. In that setting, the more general theorem produces a weaker induction hypothesis for the theorem's proof.

The approach to modular type soundness proofs presented in the remainder of this chapter follows the core idea of this technique and relies on three theorems:

FSound: a reusable *feature theorem* that is only aware of the effects that a feature uses

ESound: an *effect theorem* for a fixed set of known effects, and

LSound: a *language theorem* which combines the two to prove soundness for a specific language.

In order to maximize compatibility, the statement of the reusable feature theorem cannot hardwire the set of effects. This statement must instead rephrase type soundness in a way

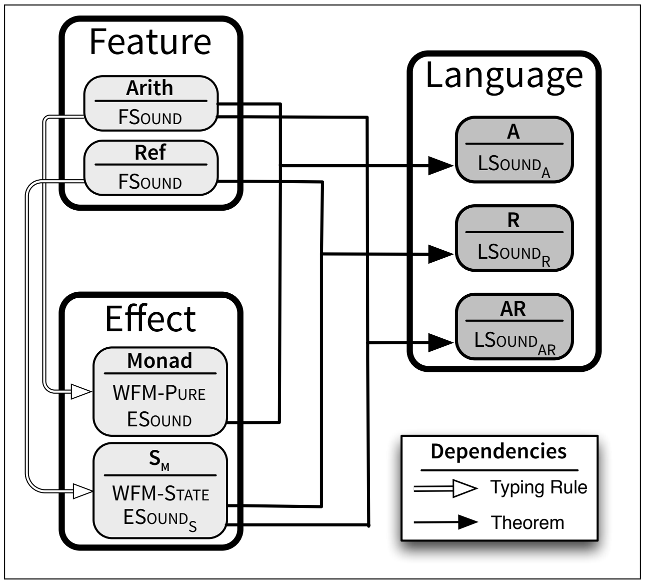


Figure 5.5: Dependency Graph

that can adapt to any superset of a feature's effects. Our solution is to have the feature theorem establish that the monadic evaluation and typing algebras of a feature satisfy an extensible well-formedness relation, defined in terms of effect-specific typing rules. Thus, a feature's proof of FSound uses only the typing rules required for the effects specific to that feature. This is akin to how features constrain the effects of the abstract semantic domains. The final language combines the typing rules of all the language's effects into a closed relation.

Figure 5.5 illustrates how these reusable pieces fit together. Each feature provides a proof algebra for FSound which relies on the typing rules (WFM-X) for the effects it uses. Each unique statement of soundness for a combination of effects requires a new proof of ESound. The proof of soundness for a particular language is synthesized entirely from a single proof of ESound and a combination of proof algebras for FSound.

Note that there are several dimensions of reuse here. A feature's proof of FSound only depends on the typing rules for the effects that feature uses and can thus be used in any language which includes those typing rules. The typing rules themselves can be reused by any number of different features. ESound depends solely on a specific combination of effects and can be reused in any language which supports that unique combination, e.g. both LSound$_A$ and LSound$_{AR}$ use ESound$_{ES}$.

5.3.1 Soundness for a Pure Feature

The reusable feature theorem FSound states that $[\![\cdot]\!]$ and typeof are related by the extensible typing relation:

$$\forall e, \Sigma. \quad \Sigma \vdash_{\mathsf{M}} [\![\mathsf{e}]\!] : \mathsf{typeof\ e} \qquad \text{(FSound)}$$

Extensible Typing Relation The extensible typing relation has the form:

$$\Sigma \vdash_{\mathsf{M}} \mathsf{v_m} : \mathsf{t_m}$$

The relation is polymorphic in an environment type env and an evaluation monad type m. The parameters Σ, $\mathsf{v_m}$ and $\mathsf{t_m}$ have types env, $\mathsf{m\ Value}$ and $\mathsf{Maybe\ Type}$ respectively. The modular typing rules for this relation can impose constraints on the environment type env and monad type m. A particular language must instantiate env and m in a way that satisfies all the constraints imposed by the typing rules used in its features.

Figure 5.4 lists the two base typing rules of this relation. These do not constrain the evaluation monad and environment types and are the only rules needed for pure features. The (WFM-Illtyped) rule denotes that nothing can be said about computations ($\mathsf{m_e}$) which are ill-typed. The (WFM-Return) rule ensures that well-typed computations only yield values of the expected type. To see how the reusable theorem works for a pure feature, consider the proof of soundness for the boolean feature.

Proof Using the above two rules, we can show that FSound holds for the boolean feature. The proof has two cases. The boolean literal case is handled by a trivial application of

(WFM-RETURN). The second case, for conditionals, is more interesting[3]:

$$(\vdash_{\mathsf{M}} [\![\mathsf{e_c}]\!] \ : \ \mathsf{typeof}\ \mathsf{e_c}) \rightarrow (\vdash_{\mathsf{M}} [\![\mathsf{e_t}]\!] \ : \ \mathsf{typeof}\ \mathsf{e_t}) \rightarrow (\vdash_{\mathsf{M}} [\![\mathsf{e_e}]\!] \ : \ \mathsf{typeof}\ \mathsf{e_e}) \rightarrow$$

$$\vdash_{\mathsf{M}} \left(\begin{array}{l} \textbf{do}\ \mathsf{v_c} \leftarrow [\![\mathsf{e_c}]\!] \\ \quad \textbf{case}\ \mathsf{isBool}\ \mathsf{v_c}\ \textbf{of} \\ \quad\quad \mathsf{Just\ b} \quad \rightarrow \\ \quad\quad\quad \textbf{if}\ \mathsf{b}\ \textbf{then}\ [\![\mathsf{e_t}]\!] \\ \quad\quad\quad\quad \textbf{else}\ [\![\mathsf{e_e}]\!] \\ \quad\quad \mathsf{Nothing} \rightarrow \mathsf{stuck} \end{array}\right) : \left(\begin{array}{l} \textbf{do}\ \mathsf{t_c} \leftarrow \mathsf{typeof}\ \mathsf{e_c} \\ \quad \mathsf{t_t} \leftarrow \mathsf{typeof}\ \mathsf{e_t} \\ \quad \mathsf{t_e} \leftarrow \mathsf{typeof}\ \mathsf{e_e} \\ \quad \mathsf{guard}\ (\mathsf{isTBool}\ \mathsf{t_c}) \\ \quad \mathsf{guard}\ (\mathsf{eqT}\ \mathsf{t_t}\ \mathsf{t_e}) \\ \quad \mathsf{return}\ \mathsf{t_t} \end{array}\right) \quad \text{(WFM-IF-VC)}$$

Because $[\![\cdot]\!]$ and typeof are polymorphic in the monad, we cannot directly inspect the values they produce. We can, however, perform case analysis on the derivations of the proofs produced by the induction hypothesis that the subexpressions are well-formed, $\vdash_{\mathsf{M}} [\![\mathsf{e_c}]\!] : \mathsf{typeof}\ \mathsf{e_c}$, $\vdash_{\mathsf{M}} [\![\mathsf{e_t}]\!] : \mathsf{typeof}\ \mathsf{e_t}$, and $\vdash_{\mathsf{M}} [\![\mathsf{e_e}]\!] : \mathsf{typeof}\ \mathsf{e_e}$. The final rule used in each derivation determines the shape of the monadic value produced by $[\![\cdot]\!]$ and typeof. Assuming that only the pure typing rules of Figure 5.4 are used for the derivations, we can divide the proof into two cases depending on whether $\mathsf{e_c}$, $\mathsf{e_t}$, or $\mathsf{e_e}$ was typed with (WFM-ILLTYPED).

- If any of the three derivations uses (WFM-ILLTYPED), the result of typeof is fail. As fail is the zero of the typing monad, (WFM-ILLTYPED) resolves the case.
- Otherwise, each of the subderivations was built with (WFM-RETURN) and the eval-

[3]We omit the environment Σ to avoid clutter.

uation and typing expressions can be simplified using the $\mathsf{return\_bind}$ monad law.

$$\vdash_{\mathsf{M}} \left(\begin{array}{l} \textbf{case}\ \mathsf{isBool}\ \mathsf{v_c}\ \textbf{of} \\ \quad \mathsf{Just\ b} \quad \rightarrow \\ \qquad \textbf{if}\ \mathsf{b}\ \textbf{then}\ \mathsf{return}\ \mathsf{v_t} \\ \qquad\qquad \textbf{else}\ \mathsf{return}\ \mathsf{v_e} \\ \quad \mathsf{Nothing} \rightarrow \mathsf{stuck} \end{array} \right) : \left(\begin{array}{l} \textbf{do}\ \mathsf{guard}\ (\mathsf{isTBool}\ \mathsf{t_c}) \\ \quad \mathsf{guard}\ (\mathsf{eqT}\ \mathsf{t_t}\ \mathsf{t_e}) \\ \quad \mathsf{return}\ \mathsf{t_t} \end{array} \right)$$

After simplification, the typing expression has reduced the bind, leaving explicit values which can be reasoned over. If $\mathsf{isTBool}\ \mathsf{t_c}$ is false, then the typing expression reduces to fail and well-formedness again follows from the WFM-ILLTYPED rule. Otherwise $\mathsf{t_c} \equiv \mathsf{TBool}$, and we can apply the inversion lemma:

$$\vdash \mathsf{v} : \mathsf{TBool} \rightarrow \exists \mathsf{b}.\ \mathsf{isBool}\ \mathsf{v} \equiv \mathsf{Just\ b}$$

to establish that $\mathsf{v_c}$ is of the form $\mathsf{Just\ b}$, reducing the evaluation to either $\mathsf{return}\ \mathsf{v_e}$ or $\mathsf{return}\ \mathsf{v_t}$. A similar case analysis on $\mathsf{eqT}\ \mathsf{t_t}\ \mathsf{t_e}$ will either produce fail or $\mathsf{return}\ \mathsf{t_t}$. The former is trivially true, and both $\vdash_{\mathsf{M}} \mathsf{return}\ \mathsf{v_t} : \mathsf{return}\ \mathsf{t_t}$ and $\vdash_{\mathsf{M}} \mathsf{return}\ \mathsf{v_e} : \mathsf{return}\ \mathsf{t_t}$ hold in the latter case from the induction hypotheses.

Modular Sublemmas The above proof assumed that only the pure typing rules of Figure 5.4 were used to type the subexpressions of the **if** expression, which is clearly not the case when the boolean feature is included in an effectful language. Instead, case analyses are performed on the extensible typing relation in order to make the boolean feature theorem compatible with new effects. Case analyses over the extensible $\vdash_{\mathsf{M}}$ relation are accomplished using extensible proof algebras which are folded over the derivations provided by the induction hypothesis, as outlined in Section 4.3.

In order for the boolean feature's proof of FSOUND to be compatible with a new effect, each extensible case analysis requires a proof algebra for the new typing rules the effect introduces to the $\vdash_{\mathsf{M}}$ relation. These proof algebras are yet another example of feature interactions. As before, these proof algebras do not need to be provided up front when developing the boolean algebra, but can instead be modularly resolved by a separate feature for the interaction of booleans and the new effect.

The formulation of the properties proved by extensible case analysis has an impact on modularity. The conditional case of the previous proof can be dispatched by folding a proof algebra for the property WFM-IF-VC over $\vdash_{\mathsf{M}} \ [\![\mathsf{v_c}]\!] \ : \ \mathsf{typeof}\ \mathsf{t_c}$. Each new effect induces a new case for this proof algebra, however, resulting in an interaction between booleans and every effect. WFM-IF-VC is specific to the proof of FSOUND in the boolean feature; proofs of FSOUND for other features require different properties and thus different proof algebras. Relying on such specific properties can lead to a proliferation of proof obligations for each new effect.

Alternatively, the boolean feature can apply a proof of a stronger property which is also applicable in other proofs, cutting down on the number of feature interactions. One such more general sublemma is WFM-BIND. This lemma shows how a proof that the monadic bind $\mathsf{v_m} \ggg\!= \mathsf{k_v}$ is well-formed follows from the well-formedness of the bound monad $\mathsf{v_m}$ and a proof that the continuation $\mathsf{k_v}\ \mathsf{v}$ is well-formed for all well-formed values v.

$$
\begin{aligned}
&(\Sigma \vdash_{\mathsf{M}} \ \mathsf{v_m} : \mathsf{t_m}) \rightarrow \\
&\qquad\qquad (\forall \mathsf{v}\ \mathsf{T}\ \Sigma' \supseteq \Sigma.\ (\Sigma' \vdash \ \mathsf{v} : \mathsf{T}) \rightarrow \ \Sigma' \vdash_{\mathsf{M}} \ \mathsf{k_v v} \ : \ \mathsf{k_t T}) \rightarrow \\
&\qquad\qquad\qquad\qquad \Sigma \vdash_{\mathsf{M}} \ \mathsf{v_m} \ggg\!= \mathsf{k_v} \ : \ \mathsf{t_m} \ggg\!= \mathsf{k_t} \quad (\text{WFM-BIND})
\end{aligned}
$$

A proof of WFM-IF-VC follows directly from two applications of this stronger property. The advantage of WFM-BIND is clear: it can be reused to deal with case analyses in other proofs of FSOUND, while a proof of WFM-IF-VC has only a single use. The disadvantage is that WFM-BIND may not hold for some new effect for which the weaker WFM-IF-VC does, possibly excluding some feature combinations. As WFM-BIND is a desirable property for typing rules, the case study focuses on that approach.

$$\frac{}{\Sigma \vdash_{\mathsf{M}} \mathsf{throw}\ \mathsf{x}\ :\ \mathsf{t_m}} \quad \text{(WFM-Throw)}$$

$$\frac{\Sigma \vdash_{\mathsf{M}} \mathsf{m} \gg\!= \mathsf{k}\ :\ \mathsf{t_m} \qquad \forall\ \Sigma' \supseteq \Sigma\ \mathsf{x}\ .\ \Sigma' \vdash_{\mathsf{M}} \mathsf{h}\ \mathsf{x} \gg\!= \mathsf{k}\ :\ \mathsf{t_m}}{\Sigma \vdash_{\mathsf{M}} \mathsf{catch}\ \mathsf{m}\ \mathsf{h} \gg\!= \mathsf{k}\ :\ \mathsf{t_m}} \quad \text{(WFM-Catch)}$$

Figure 5.6: Typing rules for exceptional monadic values.

5.3.2 Type Soundness for a Pure Language

The second phase of showing type soundness is to prove a statement of soundness for a fixed set of effects. For pure effects, the soundness statement is straightforward:

$$\forall \mathsf{v_m}\ \mathsf{t}.\ \vdash_{\mathsf{M}} \mathsf{v_m} : \mathsf{return}\ \mathsf{t} \Rightarrow \exists \mathsf{v}.\mathsf{v_m} \equiv \mathsf{return}\ \mathsf{v} \wedge \vdash \mathsf{v} : \mathsf{t} \qquad (\text{ESound}_P)$$

Each effect theorem is proved by induction over the derivation of $\vdash_{\mathsf{M}} \mathsf{v_m} : \mathsf{return}\ \mathsf{t}$. ESound$_P$ fixes the irrelevant environment type to the type () and the evaluation monad to the pure monad $\mathbb{I}$. Since the evaluation monad is fixed, the proof of ESound$_P$ only needs to consider the pure typing rules of Figure 5.4. The proof of the effect theorem is straightforward: WFM-Illtyped could not have been used to derive $\vdash_{\mathsf{M}} \mathsf{v_m} : \mathsf{return}\ \mathsf{t}$, and WFM-Return provides both a witness for v and a proof that it is of type t.

The statement of soundness for a pure arithmetic expression language is similar to ESound$_P$:

$$\forall \mathsf{e}, \mathsf{t}.\mathsf{typeof}\ \mathsf{e} \equiv \mathsf{return}\ \mathsf{t} \Rightarrow \exists \mathsf{v}.[\![\mathsf{e}]\!] \equiv \mathsf{return}\ \mathsf{v} \wedge \vdash \mathsf{v} : \mathsf{t} \qquad (\text{LSound}_{\mathsf{A}})$$

The proof of LSound$_{\mathsf{A}}$ is an immediate consequence of the reusable proofs of FSound and ESound$_P$. Folding a proof algebra for FSound over e provides a proof of $\vdash_{\mathsf{M}} [\![\mathsf{e}]\!] : \mathsf{return}\ \mathsf{t}$, satisfying the first assumption of ESound$_P$. LSound$_{\mathsf{A}}$ follows immediately. Identitical proofs hold for languages built from other combinations of pure features.

5.3.3 Errors

The evaluation algebra of the error language feature uses the side effects of the exception monad, requiring new typing rules.

Typing Rules Figure 5.6 lists the typing rules for monadic computations involving exceptions. WFM-Throw states that $\mathsf{throw}\ \mathsf{x}$ is typeable with any type. WFM-Catch states that binding the results of both branches of a catch statement will produce a monad with the same type. While it may seem odd that this rule is formulated in terms of a continuation $\ggg\!=\mathsf{k}$, it is essential for compatibility with the proofs algebras required by other features. As described in Section 5.3.1, extensible proof algebras over the typing derivation will now need cases for the two new rules. To illustrate this, consider the proof algebra for the general purpose WFM-Bind property. This algebra requires a proof of:

$$(\Sigma \vdash_{\mathsf{M}} \mathsf{catch\ e\ h} \ggg\!= \mathsf{k} \ :\ \mathsf{t_m}) \rightarrow (\forall\ \mathsf{v}\ \mathsf{T}\ \Sigma' \supseteq \Sigma.\ (\Sigma' \vdash\ \mathsf{v}\ :\ \mathsf{T}) \rightarrow\ \Sigma' \vdash_{\mathsf{M}}\ \ \mathsf{k_v}\ \mathsf{v}\ :\ \mathsf{k_t}\ \mathsf{T}) \rightarrow$$
$$\Sigma \vdash_{\mathsf{M}} (\mathsf{catch\ e\ h} \ggg\!= \mathsf{k}) \ggg\!= \mathsf{k_v} : \mathsf{t_m} \ggg\!= \mathsf{k_t}$$

With the continuation, we can first apply the associativity law to reorder the binds so that WFM-Catch can be applied: $(\mathsf{catch\ e\ h} \ggg\!= \mathsf{k}) \ggg\!= \mathsf{k_v} = \mathsf{catch\ e\ h} \ggg\!= (\mathsf{k} \ggg\!= \mathsf{k_v})$. The two premises of the rule follow immediately from the inductive hypothesis of the lemma, finishing the proof. Without the continuation, the proof statement only binds $\mathsf{catch\ e\ h}$ to $\mathsf{v_m}$, leaving no applicable typing rules.

Effect Theorem The effect theorem, ESound$_E$, for a language whose only effect is exceptions reflects that the evaluation function is either a well-typed value or an exception.

$$\forall \mathsf{v_m}\ \mathsf{t}.\ \vdash_{\mathsf{M}} \mathsf{v_m} : \mathsf{return\ t} \Rightarrow \exists x.\mathsf{v_m} \equiv \mathsf{throw}\ x \vee \exists \mathsf{v}.\mathsf{v_m} \equiv \mathsf{return\ v} \wedge \vdash \mathsf{v} : \mathsf{t} \quad (\text{ESound}_E)$$

The proof of ESound$_E$ is again by induction on the derivation of $\vdash_{\mathsf{M}} \mathsf{v_m} : \mathsf{return\ t}$. The irrelevant environment can be fixed to (), while the evaluation monad is the exception

$$\frac{\forall \sigma, \Sigma \vdash \sigma \;\rightarrow\; \Sigma \vdash_{\mathsf{M}} \mathsf{k}\ \sigma : \mathsf{t_m}}{\Sigma \vdash_{\mathsf{M}} \mathsf{get} \ggg\!= \ \mathsf{k} \ : \ \mathsf{t_m}} \quad \text{(WFM-GET)}$$

$$\frac{\Sigma' \vdash \sigma \qquad \Sigma' \supseteq \Sigma \qquad \Sigma' \vdash_{\mathsf{M}} \mathsf{k} : \mathsf{t_m}}{\Sigma \vdash_{\mathsf{M}} \mathsf{put}\ \sigma \gg \mathsf{k} \ : \ \mathsf{t_m}} \quad \text{(WFM-PUT)}$$

Figure 5.7: Typing rules for stateful monadic values.

monad $\mathbb{E}_{\mathrm{T}} \times \mathbb{I}$.

The typing derivation is built from four rules: the two pure rules from Figure 5.4 and the two exception rules from Figure 5.6. The case for the two pure rules is effectively the same as before, and WFM-THROW is straightforward. In the remaining case, $\mathsf{v_m} \equiv \mathsf{catch}\ \mathsf{e}'\ \mathsf{h}$, and we can leverage the fact that the evaluation monad is fixed to conclude that either $\exists \mathsf{v}.\mathsf{e}' \equiv \mathsf{return}\ \mathsf{v}$ or $\exists \mathsf{x}.\mathsf{e}' \equiv \mathsf{throw}\ \mathsf{x}$. In the former case, $\mathsf{catch}\ \mathsf{e}'\ \mathsf{h}$ can be reduced using catch_return, and the latter case is simplified using $\mathsf{catch\_throw}_1$. In both cases, the conclusion then follows immediately from the assumptions of WFM-CATCH. The proof of the language theorem LSOUND$_E$ is easily built from ESOUND$_E$ and FSOUND.

5.3.4 References

Typing Rules Figure 5.7 lists the two typing rules for stateful computations. To understand the formulation of these rules, consider LSOUND$_S$, the statement of soundness for a language with a stateful evaluation function. The statement accounts for both the typing environment Σ and evaluation environment σ by imposing the invariant that σ is well-formed with respect to Σ. FSOUND, however, has no such conditions (which would be anti-modular in any case). We avoid this problem by accounting for the invariant in the typing rules themselves:

- WFM-GET requires that the continuation k of a get is well-typed under the invariant.
- WFM-PUT requires that any newly installed environment maintains this invariant.

The intuition behind these premises is that effect theorems will maintain these invariants in order to apply the rules.

Effect Theorem The effect theorem for mutable state proceeds again by induction over the typing derivation. The evaluation monad is fixed to $\mathbb{S}_{\mathrm{T}}$ Sigma $\mathbb{I}$ and the environment type is fixed to $[\mathsf{Type}]$ with the obvious definitions for $\supseteq$.

- The proof case for the two pure rules is again straightforward.
- For WFM-Get we have that $\mathsf{put}\ \sigma \gg [\![\mathsf{e}]\!] \equiv \mathsf{put}\ \sigma \gg \mathsf{get} \gg\!= \mathsf{k}$. After reducing this to $\mathsf{k}\ \sigma$ with the put_get law, the result follows immediately from the rule's assumptions.
- Similarly, for WFM-Put we have that $\mathsf{put}\ \sigma \gg [\![\mathsf{e}]\!] \equiv \mathsf{put}\ \sigma \gg \mathsf{put}\ \sigma' \gg \mathsf{k}$. After reducing this to $\mathsf{put}\ \sigma' \gg \mathsf{k}$ with the put_put law, the result again follows immediately from the rule's assumptions.

5.3.5 Lambda

The case study represents the binders of the lambda feature using PHOAS [Chl08] to avoid many of the boilerplate definitions and proofs about term well-formedness found in first-order representations.

The Environment Effect Using monads allows us to represent the variable environment of the evaluation function with a reader monad $\mathbb{R}_{\mathrm{M}}$. This new effect introduces the two new typing rules listed in Figure 5.8. Unsurprisingly, these typing rule are similar to those of Figure 5.7. The rule for ask is essentially the same as WFM-Get. The typing rule for local differs slightly from WFM-Put. Its first premise ensures that whenever f is applied to an environment that is well-formed in the original typing environment Γ, the resulting environment is well-formed in some new environment Γ'. The second premise ensures the body of local is well-formed in this environment according to some type T, and the final

$$\frac{\forall \gamma.\ \Gamma \vdash \gamma\ \rightarrow\ \Gamma \vdash_{\mathsf{M}} \mathsf{k}\ \gamma : \mathsf{t_m}}{\Gamma \vdash_{\mathsf{M}} \mathsf{ask} \gg\!= \ \mathsf{k}\ :\ \mathsf{t_m}} \quad (\text{WFM-Ask})$$

$$\frac{\begin{array}{c}\forall\ \gamma.\ \Gamma \vdash \gamma \rightarrow \Gamma' \vdash \mathsf{f}\ \gamma \qquad \Gamma' \vdash_{\mathsf{M}} \mathsf{m}\ :\ \mathsf{return}\ \ \mathsf{t'_m} \\ \forall\ \mathsf{v}.\ \vdash \mathsf{v}\ :\ \mathsf{t'_m} \rightarrow \Gamma \vdash_{\mathsf{M}} (\mathsf{k}\ \mathsf{v}) : \mathsf{t_m}\end{array}}{\Gamma \vdash_{\mathsf{M}} \mathsf{local}\ \mathsf{f}\ \mathsf{m} \gg\!= \mathsf{k}\ :\ \mathsf{t_m}} \quad (\text{WFM-Local})$$

Figure 5.8: Typing rules for environment and failure monads.

$$\frac{}{\Gamma \vdash_{\mathsf{M}} \bot\ :\ \mathsf{t_m}} \quad (\text{WFM-Bot})$$

Figure 5.9: Typing rules for the failure monad.

premise ensures that k is well-formed when applied to any value of type T. The intuition behind binding the local expression in some k is the same as with put.

Modelling Non-Termination The lambda feature also introduces the possibility of non-terminating evaluation. Section 4.4 showed how evaluation can be modelled by combining *mixin algebras* with a bounded fixpoint function which applies an algebra a bounded number of times, returning a ⊥ value when the bound is exceeded. In the monadic setting, ⊥ can be captured with the fail primitive of the failure monad. This allows terminating features to be completely oblivious to whether a bounded or standard fold is used for the evaluation function, resulting in a much cleaner semantics. WFM-Bot allows ⊥ to have any type.

5.4 Effect Compositions

As we have seen, laws are essential for proofs of FSound. The proofs so far have involved only one effect and the laws regulate the behavior of that effect's primitive operations. Languages often involve more than one effect, so the proofs of effect theorems must reason about the interaction between multiple effects. There is a trade-off between fully instantiating the monad for the language as we have done previously, and continuing to reason about a constrained polymorphic monad. The former is easy for reasoning, while the latter allows the same language proof to be instantiated with different implementations of the monad. In the latter case, additional *effect interaction* laws are required.

5.4.1 Languages with State and Exceptions

Consider the effect theorem which fixes the evaluation monad to support exceptions and state. The statement of the theorem mentions both kinds of effects by requiring evaluation to be initialized with a well-formed state σ and by concluding that well-typed expressions either throw an exception or return a value. The WFM-Catch case of this theorem has the following goal:

$$(\Sigma \vdash \sigma \ : \ \Sigma) \rightarrow$$
$$\exists\ \Sigma', \sigma', \mathsf{v}. \left\{ \begin{array}{c} \mathsf{put}\ \sigma \gg \mathsf{catch\ e\ h} \ggg\!= \mathsf{k} \equiv \mathsf{put}\ \sigma' \gg \mathsf{return\ v} \\ \Sigma' \vdash \mathsf{v} : \mathsf{t} \end{array} \right\}$$
$$\vee$$
$$\exists\ \Sigma', \sigma', \mathsf{x}. \left\{ \begin{array}{c} \mathsf{put}\ \sigma \gg \mathsf{catch\ e\ h} \ggg\!= \mathsf{k} \equiv \mathsf{put}\ \sigma' \gg \mathsf{throw\ x} \\ \Sigma' \vdash \sigma' \ : \ \Sigma' \end{array} \right\}$$

In order to apply the induction hypothesis to e and h, we need to precede them by a $\mathsf{put}\ \sigma$. Hence, $\mathsf{put}\ \sigma$ must be pushed under the catch statement through the use of a law governing the behavior of put and catch. There are different choices for this law, depending on the monad that implements both $\mathbb{S}_{\mathrm{M}}$ and $\mathbb{E}_{\mathrm{M}}$. We consider two reasonable choices, based on the monad transformer compositions $\mathbb{E}_{\mathrm{T}}\ \mathsf{x}\ (\mathbb{S}_{\mathrm{T}}\ \mathsf{s}\ \mathbb{I})$ and $\mathbb{S}_{\mathrm{T}}\ \mathsf{s}\ (\mathbb{E}_{\mathrm{T}}\ \mathsf{x}\ \mathbb{I})$:

- Either catch passes the current state from when the error is thrown into the handler:
  $\mathsf{put}\ \sigma \gg \mathsf{catch\ e\ h} \equiv \mathsf{catch}\ (\mathsf{put}\ \sigma \gg \mathsf{e})\ \mathsf{h}$
- Or catch runs the handler with the initial state:
  $\mathsf{put}\ \sigma \gg \mathsf{catch\ e\ h} \equiv \mathsf{catch}\ (\mathsf{put}\ \sigma \gg \mathsf{e})\ (\mathsf{put}\ \sigma \gg \mathsf{h})$

$$\forall \Sigma, \Gamma, \delta, \gamma, \sigma, \mathsf{e_E}, \mathsf{e_T}. \left\{ \begin{array}{c} \gamma, \delta \vdash \ \mathsf{e_E} \equiv \mathsf{e_T} \\ \Sigma \vdash \sigma \ : \ \Sigma \\ \Sigma \vdash \gamma \ : \ \Gamma \\ \mathsf{typeof}\ \mathsf{e_T} \equiv \mathsf{return\ t} \end{array} \right\} \rightarrow$$

$$\exists\ \Sigma', \sigma', \mathsf{v}. \left\{ \begin{array}{c} \mathsf{local}\ (\lambda\_.\gamma)\ (\mathsf{put}\ \sigma \gg [\![\mathsf{e_E}]\!]) \equiv \mathsf{local}\ (\lambda\_.\gamma)\ (\mathsf{put}\ \sigma' \gg \mathsf{return\ v}) \\ \Sigma' \vdash \mathsf{v : t} \end{array} \right\}$$

$$\vee$$

$$\exists\ \Sigma', \sigma', \mathsf{v}. \left\{ \begin{array}{c} \mathsf{local}\ (\lambda\_.\gamma)\ (\mathsf{put}\ \sigma \gg [\![\mathsf{e_E}]\!]) \equiv \mathsf{local}\ (\lambda\_.\gamma(\mathsf{put}\ \sigma' \gg \bot) \\ \Sigma' \vdash \sigma' \ : \ \Sigma' \\ \Sigma' \supseteq \Sigma \end{array} \right\}$$

$$\vee$$

$$\exists\ \Sigma', \sigma', \mathsf{v}. \left\{ \begin{array}{c} \mathsf{local}\ (\lambda\_.\gamma)\ (\mathsf{put}\ \sigma \gg [\![\mathsf{e_E}]\!]) \equiv \mathsf{local}\ (\lambda\_.\Gamma(\mathsf{put}\ \sigma' \gg \mathsf{throw\ t}) \\ \Sigma' \vdash \sigma' \ : \ \Sigma' \\ \Sigma' \supseteq \Sigma \end{array} \right\}$$

$$(\text{ES}\textsc{ound}_{ESRF})$$

Figure 5.10: Effect theorem statement for errors, state, an environment and failure.

The WFM-Catch case is provable under either choice. As the LSound$_{ES}$ proof is written as an extensible theorem, the two cases are written as two separate proof algebras, each with a different assumption about the behavior of the interaction. Since the cases for the other rules are impervious to the choice, they can be reused with either proof of WFM-Catch.

5.4.2 Full Combination of Effects

A language with references, errors and lambda abstractions features four effects: state, exceptions, an environment and failure. The language theorem for such a language relies on the effect theorem ESound$_{ESRF}$ given in Figure 5.10. The proof of ESound$_{ESRF}$ is similar to the previous effect theorem proofs, and makes use of the full set of interaction laws given in Figure 5.11. Perhaps the most interesting observation here is that because the environment monad only makes local changes, we can avoid having to choose between laws regarding how it interacts with exceptions. Also note that since we are representing nontermination using a failure monad FailMonad m, the catch_fail law conforms to our desired semantics.

Exceptional Environment

```
class (𝔼_M x m, MonadEnvironment m) ⇒ MonadErrorEnvironment x g m where
  local_throw :: local f (throw e) ≡ throw e
  local_catch :: local f (catch e h) ≡
                   catch (local f e) (λx. local f (h x))
```

Exceptional Failure

```
class (𝔼_M x m, FailMonad m) ⇒ MonadFailState x m where
  catch_fail      :: catch fail h ≡ fail
  fail_neq_throw :: fail ≢ throw x
```

Exceptional State Failure

```
class (𝔼_M x m, 𝕊_M s m, FailMonad m) ⇒ MonadFailErrorState x m where
  put_fail_throw :: put σ ≫ fail ≢ put σ′ ≫ throw x
```

Exceptional State

```
class (𝔼_M x m, FailMonad m) ⇒ MonadErrorState x m where
  put_ret_throw :: put σ ≫ return a ≢ put σ′ ≫ throw x
  put_throw :: ∀A B.put σ ≫ throw_A x ≡ put σ′ ≫ throw_A x →
                put σ ≫ throw_B x ≡ put σ′ ≫ throw_B x
```

Alternate Exceptional State laws

```
class (𝔼_M x m, FailMonad m) ⇒ MonadErrorState_1 x m where
  put_catch_1 :: put σ ≫ catch e h ≡ catch (put σ ≫ e) h
```

Or

```
class (𝔼_M x m, FailMonad m) ⇒ MonadErrorState_2 x m where
  put_catch_2 :: put env ≫ catch e h ≡
                   catch (put σ ≫ e) (λx → put σ ≫ h x)
```

Figure 5.11: Interaction laws

5.5 Case Study

Using monads allows us to extend the case study of the previous chapter with effectful features, enhancing the mini-ML variant [CDKD86] variant with references and errors. The study builds twenty eight different combinations of features which are all legal combinations with at least one feature providing values. Figure 5.12 presents the syntax of the expressions, values, and types provided; each line is annotated with the feature that provides that set of definitions.

$\mathtt{e} ::= \mathbb{N} \mid \mathtt{e + e}$	*Arith*
$\mid \mathbb{B} \mid$ **if** $\mathtt{e}$ **then** $\mathtt{e}$ **else** $\mathtt{e}$	*Bool*
$\mid$ **lam** $\mathtt{x : T.e} \mid \mathtt{e\ e} \mid \mathtt{x}$	*Lambda*
$\mid$ **ref** $\mathtt{e} \mid \mathtt{!e} \mid \mathtt{e{:}{=}e}$	*References*
$\mid$ **try** $\mathtt{e}$ **with** $\mathtt{e} \mid$ **error**	*Errors*

$\mathtt{V} ::= \mathbb{N}$	*Arith*
$\mid \mathbb{B}$	*Bool*
$\mid$ **clos** $\mathtt{e}\ \overline{\mathtt{V}}$	*Lambda*
$\mid$ **loc** $\mathbb{N}$	*References*

$\mathtt{T} ::=$ **Nat**	*Arith*
$\mid$ **Bool**	*Bool*
$\mid \mathtt{T} \rightarrow \mathtt{T}$	*Lambda*
$\mid$ **Ref** $\mathtt{T}$	*References*

Figure 5.12: Expressions, values, and types used in the case study.

Four kinds of feature interactions appear in the case study.

- The PHOAS representation of binders requires an auxiliary equivalence relation, per the previous section. The soundness proofs of language theorems for languages which include binders proceed by induction over this equivalence relation instead of expressions. The reusable feature theorems of other features need to be lifted to this equivalence relation.

- The effect theorems that feature an environment typing Σ, such as those for state or environment, need a weakening sublemma which states that each well-formed value under Σ is also well-formed under a conservative extension:

$$\Sigma \vdash v \ : \ t \rightarrow \Sigma' \supseteq \Sigma \rightarrow \Sigma' \vdash v \ : \ t$$

- Inversion lemmas for the well-formed value relation as in the proof of FSound for the boolean feature in Section 5.3.1 are proven by induction over the relation.

The proofs of the first and second kind of feature interactions are straightforward;

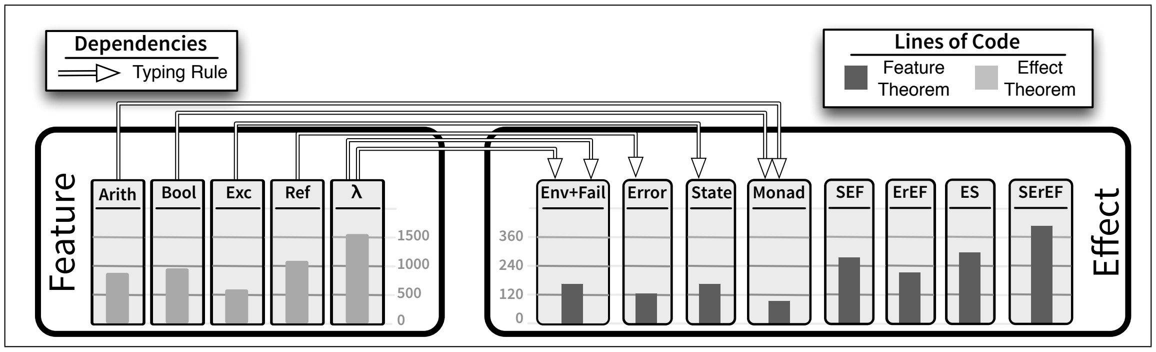


Figure 5.13: Dependency and size information for the features and effects used in the case study.

the inversion lemmas of the third kind can again be dispatched by tactics hooked into the type class inference algorithm.

The framework itself consists of about 4,400 LoC of which about 2,000 LoC comprise the implementation of the monad transformers and their algebraic laws. The size in LoC of the implementation of semantic evaluation and typing functions and the reusable feature theorem for each language feature is given in the left box in Figure 5.13. The right box lists the sizes of the effect theorems. Each language needs on average 110 LoC to assemble its semantic functions and soundness proofs from those of its features and the effect theorem for its set of effects.

Chapter 6

Safe Composition

The previous chapters have shown how to achieve feature modularity in the domains of software product lines through an extension to Java and mechanized metatheory through a different formulation of datatypes and semantic domains. While the techniques were different, the benefits were similar. For software product lines, Chapter 2 showed how type safety of a program built from LFJ features can be derived from the type safety of those features. For mechanized meta-theory, Chapters 4 and 5 demonstrated how proofs of type safety for a language built as a composition of functors could be built from modular proof algebras over those functors. In both cases, proper feature modularity allows proofs of properties of individual compositions of features to be derived from independently-developed proofs about the included features. This chapter shows how feature modularity can be exploited to efficiently reason about an entire family of compositions built from a common set of features.

Safe Composition of a property π for a product line is defined as π holding for all the members of the family. In software product lines, for example, every valid product specification should produce a program without any type errors. Similarly, a family of programming languages should all have sound type systems. Safe composition of a desired property π of a syntactic mapping δ for a feature model FM describing the valid selections

$\mathcal{P}$ of a set of features $\mathcal{F}$ is formally expressed as:

$$\forall\, \mathcal{P} \subseteq \mathcal{F}, \mathsf{FM}(\mathcal{P}) = \mathsf{true} \rightarrow \pi \left(\sum_{\mathsf{F} \in \delta(\mathcal{P})} \mathsf{F} \right) \tag{6.1}$$

A naive approach to establishing safe composition is to simply generate each feature selection $\mathcal{P}$ allowed by FM and individually prove that $\pi(\sum_{\mathsf{F} \in \delta(\mathcal{P})} \mathsf{F})$ holds. The number of possible members of a product line grows exponentially in the number of features, making checking each individual composition inefficient even for properties which can be established through automated analyses such as type-checking. Proper feature modularity of the product line as detailed in Section 1.2.3 yields two immediate benefits:

1. The proof of $\pi(\sum_{\mathsf{F} \in \delta(\mathcal{P})} \mathsf{F})$ for a feature selection $\mathcal{P}$ can be reduced to a composition of semantic modules $\sum_{\mathsf{F} \in \rho(\mathcal{P})} \mathsf{F}$, so it is no longer necessary to generate the entire set of individual products for analysis.

2. Semantic feature modules F_ρ establishing π for syntactic feature modules F_δ can be built once and for all and then reused for specific compositions of syntactic feature modules. In the alternate approach, the exports of F_δ need to be reanalyzed in the context of each composition in which F_δ is included.

While composing semantic feature modules together is an improvement over generating and reasoning about each composition, it still must wrestle with the combinatorics of composing modules for each possible product. In order to discuss efficient approaches to checking safe composition for a broad set of domains, the following section formalizes a generic syntactic and semantic feature module system.

6.1 A Feature Module System

In order to capture the broad set of domains that feature modules can be implemented in, the feature module system must be quite general. To align with the mappings from

v, x	Variation Points
f	Fragments
$\mathsf{m}_\delta = \{\overline{\mathsf{v}}, \overline{\mathsf{x} \mapsto \mathsf{f}}\}$	Syntactic Modules
$\mathtt{imports}(\mathsf{m}_\delta) \triangleq \overline{\mathsf{v}} = \mathsf{VP}(\overline{\mathsf{f}})$	Syntactic Module Imports
$\mathtt{exports}(\mathsf{m}_\delta) \triangleq \overline{\mathsf{x}}$	Syntactic Module Exports

$$\frac{\overline{\mathsf{v}} \cap \overline{\mathsf{x}} = \emptyset \qquad \mathsf{VP}(\overline{\mathsf{f}}) \subseteq \overline{\mathsf{v}} \cup \overline{\mathsf{x}} \qquad \mathtt{acyclic}(\overline{\mathsf{x} \mapsto \mathsf{f}})}{\{\overline{\mathsf{v}}, \overline{\mathsf{x} \mapsto \mathsf{f}}\}\ \mathtt{OK}}$$

$$\frac{\overline{\mathsf{x_m}} \cap \overline{\mathsf{x_n}} = \emptyset \qquad \mathtt{acyclic}(\overline{\mathsf{x_m} \mapsto \mathsf{f_m}} \cup \overline{\mathsf{x_n} \mapsto \mathsf{f_n}})}{\{\overline{\mathsf{v_m}}, \overline{\mathsf{x_m} \mapsto \mathsf{f_m}}\} +_\delta \{\overline{\mathsf{v_n}}, \overline{\mathsf{x_n} \mapsto \mathsf{f_n}}\} = \{\overline{\mathsf{v_m}} \cup \overline{\mathsf{v_m}} - (\mathsf{x_m} \cup \mathsf{x_n}),\ \overline{\mathsf{x_m} \mapsto \mathsf{f_m}} \cup \overline{\mathsf{x_n} \mapsto \mathsf{f_n}}\}}$$

Figure 6.1: Definitions for Syntactic Feature Modules

Section 1.2.2, the system has both syntactic and semantic modules[1]. Each imported and exported definition and proof is tagged with a name v.

Syntactic modules (Figure 6.1) correspond to LFJ feature modules (without typing information), or the syntax and semantics of a programming language formalization. A module m_δ is a set of named external references and a set of named fragments. A fragment f is a definition of the syntactic domain, possibly with some missing references. The auxiliary function VP returns the set of external references in a fragment. A LFJ fragment, for example, would be a class, method, field, or refinement in a LFJ feature module, and VP would be the set of externally defined classes, fields, or methods referenced in that fragment.

The import interface of such a module m_δ is just a list of names of external references (undefined variation points), and the export interface is the set of names of the exported statements. The contract enforced by module abstraction (m_δ OK) is simply that when all VPs in $\mathsf{import}(\mathsf{m}_\delta)$ are defined, the module can export its set of statements (i.e. the module is complete). Syntactic modules are composed with the $+_\delta$ operator and obey the

[1]This distinction is irrelevant in the metatheory mechanized in Coq, as there is no distinction between proofs and programs.

definition of mixin composition:

$$\begin{aligned}
\mathsf{imports}(\mathsf{m}_\delta +_\delta \mathsf{n}_\delta) = &\ \mathsf{imports}(\mathsf{m}_\delta) \cup \mathsf{imports}(\mathsf{n}_\delta) - (\mathsf{exports}(\mathsf{m}_\delta) \cup \mathsf{exports}(\mathsf{n}_\delta)) \\
\mathsf{exports}(\mathsf{m}_\delta +_\delta \mathsf{n}_\delta) = &\ \mathsf{exports}(\mathsf{m}_\delta) \cup \mathsf{exports}(\mathsf{n}_\delta)
\end{aligned}$$

$\pi_\mathsf{v}, \pi_\mathsf{x}$	Properties of Variation Points
$\rho : \pi_\mathsf{x}$	Proofs of Fragment Properties
$\mathsf{m}_\pi = \{\mathsf{m}_\delta, \overline{\mathsf{u} : \pi_\mathsf{v}}, \overline{\mathsf{w} \mapsto \rho : \pi_\mathsf{x}}\}$	Semantic Modules
$\mathsf{assumes}(\mathsf{m}_\pi) \triangleq \overline{\mathsf{u} : \pi_\mathsf{v}}$	Semantic Module Assumptions
$\mathsf{proves}(\mathsf{m}_\pi) \triangleq \overline{\mathsf{w} : \pi_\mathsf{x}}$	Semantic Module Exports

$$\frac{\begin{array}{c}\overline{\mathsf{u}} \cap \overline{\mathsf{w}} = \emptyset \qquad \overline{\overline{\rho_\mathsf{v} : \pi_\mathsf{v}} \vdash \rho_\mathsf{x} : \pi_\mathsf{x}} \qquad \mathsf{consistent}(\mathsf{w} \mapsto \rho : \pi_\mathsf{x}) \\ \overline{\mathsf{u} \in \mathsf{imports}(\mathsf{m}_\delta)} \qquad \overline{\mathsf{w} \in \mathsf{exports}(\mathsf{m}_\delta)} \qquad \mathsf{m}_\delta\ \mathsf{OK}\end{array}}{\{\mathsf{m}_\delta, \overline{\mathsf{u} : \pi_\mathsf{v}}, \overline{\mathsf{w} \mapsto \rho : \pi_\mathsf{x}}\}\ \mathsf{OK}}$$

$$\frac{w_m \cap w_n = \emptyset \qquad \mathsf{consistent}(\overline{w_m \mapsto \rho_m : \pi_{x_m}} \cup \overline{w_n \mapsto \rho_n : \pi_{x_n}})}{\begin{array}{c}\{\mathsf{m}_\delta, \overline{\mathsf{u_m} : \pi_{\mathsf{v_m}}}, \overline{\mathsf{w_m} \mapsto \rho_\mathsf{m} : \pi_{\mathsf{x_m}}}\} +_\pi \{\mathsf{n}_\delta, \overline{\mathsf{u_n} : \pi_{\mathsf{v_n}}}, \overline{\mathsf{w_n} \mapsto \rho_\mathsf{n} : \pi_{\mathsf{x_n}}}\} = \\ \{\mathsf{m}_\delta +_\delta \mathsf{n}_\delta, \overline{\mathsf{u_m} : \pi_{\mathsf{v_m}}} \cup \overline{\mathsf{u_n} : \pi_{\mathsf{v_n}}} - (\overline{\mathsf{w_m} : \pi_{\mathsf{x_m}}} \cup \overline{\mathsf{w_n} : \pi_{\mathsf{x_n}}}), \\ \overline{\mathsf{w_m} \mapsto \rho_\mathsf{m} : \pi_{\mathsf{x_m}}} \cup \overline{\mathsf{w_n} \mapsto \rho_\mathsf{n} : \pi_{\mathsf{x_n}}}\}\end{array}}$$

Figure 6.2: Definitions for Semantic Feature Modules

The interfaces of *semantic modules* (Figure 6.2) carry more information than syntactic interfaces. The properties that a semantic module m_π imports or $\mathsf{assumes}$ and the proofs the module exports or proves have both a name u and a signature of a property π of a name v (written π_v). Every semantic module m_π references a syntactic module m_δ and includes a set of named assumed properties $\overline{\mathsf{u} : \pi_\mathsf{v}}$ and a set of named exported proofs $\overline{\mathsf{w} \mapsto \rho : \pi_\mathsf{x}}$. These exported proofs ρ are of properties π_x of fragments f where $\mathsf{x} \mapsto \mathsf{f} \in \mathsf{exports}(\mathsf{m}_\delta)$. All the proofs exported by m_π are built under a set of assumptions $\overline{\pi_\mathsf{v}}$ on external references $\overline{\mathsf{v}} \in \mathsf{imports}(\mathsf{m}_\delta)$. In the case of LFJ, a semantic module exports typing derivations for the associated $\mathsf{feature}$ module: $\vdash_\tau\ md \mid \mathcal{C}_\mathsf{v}$. These proofs demonstrate that a method md is well-formed if the constraints $\mathcal{C}_\mathsf{v}$ on external references v are satisfied. The contract enforced by a well-formed semantic module (m_π OK) is if each $\mathsf{v} \in \mathsf{imports}(\mathsf{m}_\delta)$ is bound to a statement that satisfies π_v, then π_x will hold of the statements of $\overline{\mathsf{x}}$ exported by m_δ. The system used to derive this assumption depends on the properties of interest.

Syntactic modules can in fact be seen as simplified semantic modules. The definitions of Figure 6.1 can be derived from Figure 6.2 by setting m_δ to a simple base feature with a single trivial export x and only allowing trivial properties $\pi_\mathsf{x} = \mathsf{True}$.

Module Compatibility Two modules m and n are *incompatible* if any composition which contains m and n cannot be a complete, well-formed module (and thus can never export anything useful). In the case of syntactic modules, this only occurs when m_δ and n_δ define the same variation point or have cyclic definitions. The $+_\delta$ operator checks these two conditions in order to ensure that the resulting module could be compatible with other modules.

Compatibility of semantic modules is trickier. Two semantic modules m_π and n_π are incompatible if they export invalid proofs of some property π. This occurs when an exported proof is used to satisfy one of its own assumptions—akin to a cyclic reference in a syntactic module. The $+_\pi$ operator prevents this using the $\mathsf{consistent}$ predicate of Figure 6.2. The specific mechanism used to check consistency depends on the underlying proof system. Two modules are also incompatible if the assumptions of a composition containing them can never be met. The $+_\pi$ operator does not enforce this, as this cannot be checked for undecidable properties. Incompatibility *can* be shown when m_π and n_π assume contradictory properties (e.g. π_v and $\neg\pi_\mathsf{v}$). It is impossible to satisfy the assumptions of their composition $\mathsf{m}_\pi +_\pi \mathsf{n}_\pi$ because proofs of contradictory facts cannot be constructed in a consistent logic.[2]

To ensure that a module m_π may eventually prove something, it should not need to import any properties defined on its exported variation points ($\mathsf{exports}(\mathsf{m}_\pi)$). We formalize

[2]This is precisely how consistent refinement is ensured in product lines with proofs—incompatible features will inevitably demand construction of proofs of contradictory facts, which the consistency of the underlying logic will not allow.

this as the closed predicate:

$$\mathsf{closed}(\mathsf{m}_\pi) \triangleq \forall \mathsf{u} \ : \ \pi_{\mathsf{v}} \in \mathsf{assumes}(\mathsf{m}_\pi) \rightarrow \mathsf{v} \in \mathsf{imports}(\mathsf{m}_\delta)$$

We also define a complementary open predicate to describe modules which have proofs that could be discharged. This means that a closed semantic module (whose syntactic module has an empty $\mathsf{imports}$ interface) will have an empty $\mathsf{assumes}$ interface. Note that the converse does not necessarily hold.

In order to present a more general formulation of feature modules, our system diverges slightly from the standard definition of modules in which definitions are coupled with semantic interfaces. Standard modules systems, for example, export a set of program statements and proofs of type safety. By separating syntactic and semantic modules, we are able to consider several different abstractions of a set of statements according to the property of interest. Each semantic module abstracts the statements of a syntactic module in isolation. Once the abstraction is shown to hold, the guarantees of the underlying module system allow us to reason about a composition of syntactic modules using only the interfaces of the associated semantic modules. Thus, we need only check that the interfaces of semantic modules are satisfied to verify the composition of syntactic modules. It is possible to represent k independent properties as k semantic modules, a single module exporting proofs of all properties, or any combination of the two.

6.2 Scaling Semantic Composition

We now return to the question of checking safe composition for a set of features $\mathcal{F}$ in the context of this general feature module system. Checking the well-formedness of semantic feature module F_π OK statically takes a fixed amount of work, w_{F}. Assuming that it takes roughly as much work to check that some property π holds for a composition of features specified by a selection P as it does to check that property for all features of that composition individually, $\mathsf{w_P} \approx \sum_{\mathsf{F} \in \mathcal{P}} \mathsf{w_F}$. Since each optional feature F appears in $2^{|\mathcal{F}|-1}$ products, it takes $\sum_{\mathsf{F} \in \mathcal{P}} w_{\mathsf{F}} \cdot 2^{|\mathcal{F}|-1} = 2^{|\mathcal{F}|-1} \cdot \sum_{\mathsf{F} \in \mathcal{P}} w_{\mathsf{F}}$ to check safe composition of π for $\mathcal{F}$ by generating and checking each product individually.

Once the well-formedness of each semantic feature F_π has been established at a cost of $\mathsf{w_F}$, the modules still need to be composed to verify a specific product. Composing two semantic modules takes some work k, which is typically much less than checking module well-formedness, $\mathsf{k} \ll \mathsf{w_F}$. Given semantic feature modularity, a feature selection $\mathcal{P}$ can be verified at a cost of $\mathsf{k} \cdot |\mathcal{P}| - 1$. (Each product has a complementary feature selection of length $2^{|\mathcal{F}|} - |\mathcal{P}|$, so it takes $2^{|\mathcal{F}|} \cdot \mathsf{k} - 2$ to compose the semantic modules for the two complementary selections.) Figure 6.2 highlights two such complementary feature selections ($\mathsf{F} + \mathsf{G} + \mathsf{I}$ and H) in gray. Thus, it takes $|\mathcal{F}| \cdot 2^{|\mathcal{F}|} \cdot \mathsf{k} - 2^{|\mathcal{F}|} + 1$ to check the interfaces of each product in the product line. Incorporating the cost of individual analysis yields a total cost of $\sum_{\mathsf{F} \in \mathcal{P}} \mathsf{w_F} + |\mathcal{F}| \cdot 2^{|\mathcal{F}|} \cdot \mathsf{k} - 2^{|\mathcal{F}|} + 1$ to check safe composition of F using semantic module composition. This yields a roughly $\mathsf{w/k}$ speedup over the naive approach (recall that typically $\mathsf{k} \ll \mathsf{w_F}$).

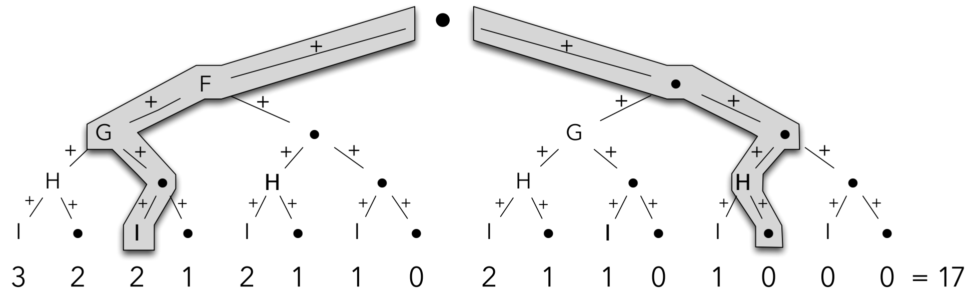


Figure 6.3: Interface checks for each product in $\sum_{\mathsf{F} \in \{\mathsf{F,G,H,I}\}} \mathsf{F}$.

This approach still requires a potentially exponential number of compositions to build intermediate semantic modules. The number of compositions can be reduced by exploiting the algebraic properties of module composition to memoizing the interfaces of composite modules. Once a composite semantic module $\mathsf{m}_\pi + \mathsf{n}_\pi$ has been built, it is not necessary recheck the compatibility of m_π and n_π or reconstruct its import and export interfaces. Since this process is independent of the surrounding modules (associativity), we can reuse this work for any composition including m_π and n_π. We can thus gain some improvement by keeping a database of the interfaces of composite modules built when checking an individual selection. When checking subsequent selections which include that composite module, the interface from that database is reused and work is avoided[3]. The possible product specifications in a family can be ordered so that each product differs from one before it by exactly one feature. This allows us to iteratively check the interface of each product specification with a single composition at each step, reducing the number of interface checks from $|\mathcal{F}| \cdot 2^{|\mathcal{F}|} \cdot \mathsf{k} - 2^{\mathsf{n}} + 1$ to $2^{|\mathcal{F}|} - |\mathcal{F}| - 1$- a speedup of $|\mathcal{F}|$ without a loss of precision. Figure 6.2 illustrates this approach.

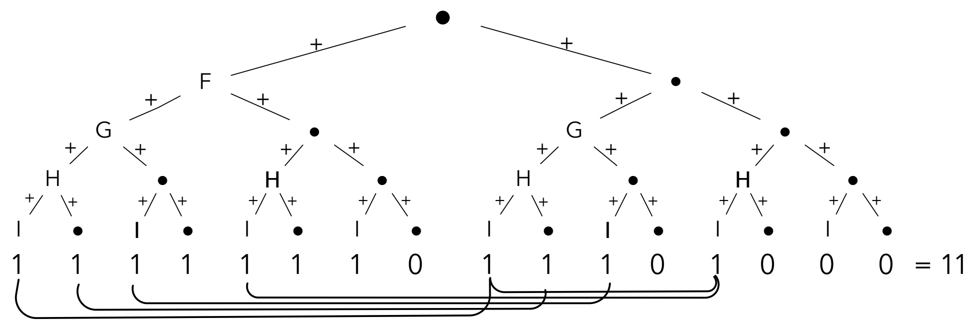


Figure 6.4: Memoized Interface checks for each product of $\{\mathsf{F}, \mathsf{G}, \mathsf{H}, \mathsf{I}\}$.

Safe Composition is NP-Hard

While exploiting feature modularity and memoizing interface composition is a substantial speedup ($O(|\mathcal{S}| \cdot \sum_{\mathsf{F} \in \mathcal{F}} \mathsf{w}_\mathsf{F})$) over a naive approach to safe composition, exploiting semantic modularity in order to check safe composition still requires an amount of work that is exponential in the number of features in a product line. Unfortunately, this speedup is

[3]If semantic module composition ever fails, we can stop— safe composition does not hold.

the best we can hope to achieve for a precise solution to the general statement of safe composition in 6.1 for non-trivial (i.e. non-tautological) properties with complex feature models.

Proof. Suppose that we have a oracle SC that, when given a a set of features $\mathcal{F}$ and feature model FM, checks safe composition of some non-trivial property π. Since π is not a tautology, there must exist some object for which the property does not hold[4]. Encode this object as a base feature module B_δ with no variability. For an arbitrary SAT instance E with variables $\overline{\mathsf{X}}$, create $|\overline{\mathsf{X}}|$ empty modules $\overline{\mathsf{X}_\delta}$ which neither export anything nor import anything. Take the feature model of this product line to be $\mathsf{FM}_\mathsf{E} = \mathsf{E} \wedge \mathsf{B}$.

Since every non-base feature is empty and B is always included, every product is equal to B_δ and π therefore does not hold for any product in the line. Thus, safe composition of this product line can only hold when there are no satisfying assignments to FM_E– e.g. the product line is empty. Since $\mathsf{FM}_\mathsf{E} \rightarrow \mathsf{E}$, we can conclude that $\mathsf{SC}(\overline{\mathsf{X}_\delta} \cup \mathsf{B}_\delta, \mathsf{E} \wedge \mathsf{B}) = \mathsf{SAT}(\mathsf{E})$. Having reduced SAT to safe composition, we can conclude the latter is NP-hard and that a sub-exponential, precise algorithm for checking safe composition of non-trivial properties most likely does not exist. □

6.3 Reducing To SAT

Just because checking safe composition is NP-Hard does not mean all hope is lost, however. Many SAT instances can be solved efficiently by modern SAT solvers, which we can leverage by reducing safe composition to satisfiability. To demonstrate this, we first consider how the constraints from Chapter 2 can be used to efficiently check safe composition of LFJ product lines.

[4]In the case of type-safety, for example, this is any ill-typed program, i.e. "fred" + 42.

6.3.1 Safe Composition of LFJ Product Lines

Safe composition of an SPL checks that all valid feature selections compose into well-formed programs. Defining $\pi(\mathcal{S})$ to be $\vdash \sum_{\delta} \mathcal{S}\ \mathsf{OK}$, the statement of safe composition for SPLs is simply an instantiation of (6.1). Having established the well-formedness of LFJ feature modules, soundly establishing safe composition is a matter of checking constraint[5] satisfaction for every possible feature selection:

Theorem 6.3.1 (Soundness of Feature Typing)**.** *If a set of features $\mathcal{F}$ is well-formed subject to a set of constraints $\mathcal{C}_{\mathsf{F}}$, and if the composition of every valid selection of features satisfies the constraints $\mathcal{C}_{\mathsf{F}}$ of its constituent features, then every valid feature selection builds a well-typed program.*

$$\{\forall\ \mathsf{F} \in \mathcal{F}, \vdash \mathsf{F}\ :\ \mathcal{C}_{\mathsf{F}}\} \rightarrow$$
$$\left\{\forall \mathcal{S} \subseteq \mathcal{F}, \mathsf{FM}(\mathcal{S}) = \mathsf{true} \rightarrow \sum_{\delta} \mathcal{S} \vDash \bigcup_{\mathsf{F}\in\mathcal{S}} \mathcal{C}_{F} \rightarrow \right\}$$
$$\forall \mathcal{S}' \subseteq \mathcal{F}, \mathsf{FM}(\mathcal{S}') = \mathsf{true} \rightarrow\ \vdash \sum_{\delta} \mathcal{S}'\ \mathsf{OK}$$

We can now check TypeSafe composition of an SPL by generating a SAT formula from the constraints generated by the LFJ type system. This formula is built by iterating over the set of constraints for each LFJ feature module and analyzing the exports of each feature in the product line to see they satisfy that constraint. A SAT clause is generated that encodes the fact that as long as one of the satisfying features is included, that constraint will be satisfied. If a constraint entails the inclusion of another feature, the boolean variable representing that feature is set to true. When checking if a given constraint is satisfied, a SAT solver will add conflict clauses encoding the possible feature selections that satisfy that constraint so that they do not have to be rechecked.

[5] i.e. import interface.

6.3.2 Building SAT Formulas

The LFJ type system checks whether a given product specification falls into the subset of type-safe specifications described by a feature table's constraints. Checking safe composition of a product line amounts to showing the feature model is contained within the subset of type-safe products. The variables used in the propositional representation of feature models have the form of the first two entries given in Figure 6.5. An assignment to the **In** and **Prec** variables which obeys the properties of the precedence relation describes a unique product specification. A product line is described by the satisfying assignments to the formula for its feature model.

$\mathbf{In}_{\mathsf{A}}$	: Feature A is included.
$\mathbf{Prec}_{\mathsf{A},\mathsf{B}}$	: Feature A precedes Feature B.
$\mathbf{Sty}_{\tau_1,\tau_2}$	: τ_1 is a subtype of τ_2.

Figure 6.5: Description of propositional variables.

These variables can describe the type-safe product specifications in propositional logic using the constraints generated by the LFJ type system. The definitions exported by a LFJ feature module are well-formed when the modules constraints are satisfied. Figure 6.6 gives the **Propify** function which translates each constraint into a propositional description of the product specifications which satisfy that constraint per the rules in Figure 2.10. Given $\vdash\ \mathsf{F} \mid \mathcal{C}_{\mathsf{F}}$, $\mathbf{Propify}(\mathcal{S}, \mathcal{C}_{\mathsf{F}}) \rightarrow \mathcal{S} \models \mathcal{C}_{\mathsf{F}}$. The set of product specifications which satisify the constraints on a feature F is simply the conjunction of those formulas, $\mathcal{C}_{\mathsf{F}}$. Thus, the set of well-typed product specifications is $\bigwedge_{\mathsf{F}} \mathbf{In}_{\mathsf{F}} \rightarrow \mathcal{C}_{\mathsf{F}}$.

Additional constraints are needed to ensure that each satisfying assignment corresponds to a valid product specification. Figure 6.7 gives the propositional formula imposing these properties. The first three formulas enforce that a precedence relation is total on all features included in a product specification, that it is asymmetric, and that it is irreflexive. The next four ensure that each product specification dictates an assignment to the **Sty** variables corresponding to its class hierarchy. The STY_TOTAL rule builds the transitive

$$
\begin{array}{ll}
\tau_1 \prec \tau_2 & \Rightarrow \mathbf{Sty}_{\tau_1,\tau_2} \\
\tau_2 \prec \mathbf{ftype}(\tau_1, \mathsf{f}) & \Rightarrow \bigvee\{\mathbf{Sty}_{\tau_2,\mathsf{cl}} \wedge \mathbf{Sty}_{\tau_1,\mathbf{type}(\mathsf{cld})} \wedge \mathbf{FinalIn}_{\mathbf{name}(\mathsf{cld}),\mathsf{F}} \mid \exists \mathsf{cld} \in \mathbf{clds}(\mathsf{F}), \exists \mathsf{cl}, \mathsf{cl\ f} \in \mathbf{fds}(\mathsf{cld})\} \vee \\
 & \quad \bigvee\{\mathbf{Sty}_{\tau_2,\mathsf{cl}} \wedge \mathbf{Sty}_{\tau_1,\mathbf{type}(\mathsf{rcld})} \wedge \mathbf{Final}_{\mathbf{name}(\mathsf{rcld}),\mathsf{F}} \mid \exists \mathsf{rcld} \in \mathbf{rclds}(\mathsf{F}), \exists \mathsf{cl}, \mathsf{cl\ f} \in \mathbf{fds}(\mathsf{rcld})\} \\
\mathbf{ftype}(\tau_1, \mathsf{f}) \prec \tau_2 & \Rightarrow \bigvee\{\mathbf{Sty}_{\mathsf{cl},\tau_2} \wedge \mathbf{Sty}_{\tau_1,\mathbf{type}(\mathsf{cld})} \wedge \mathbf{FinalIn}_{\mathbf{name}(\mathsf{cld}),\mathsf{F}} \mid \exists \mathsf{cld} \in \mathbf{clds}(\mathsf{F}) \exists \mathsf{cl}, \mathsf{cl\ f} \in \mathbf{fds}(\mathsf{cld})\} \vee \\
 & \quad \bigvee\{\mathbf{Sty}_{\mathsf{cl},\tau_2} \wedge \mathbf{Sty}_{\tau_1,\mathbf{type}(\mathsf{rcld})} \wedge \mathbf{Final}_{\mathbf{name}(\mathsf{rcld}),\mathsf{F}} \mid \exists \mathsf{rcld} \in \mathbf{rclds}(\mathsf{F}), \exists \mathsf{cl}, \mathsf{cl\ f} \in \mathbf{fds}(\mathsf{rcld})\} \\
\mathbf{mtype}(\tau, \mathsf{m}) \prec \overline{\pi_\mathsf{k}}^\mathsf{k} \to \pi & \Rightarrow \bigvee\{\mathbf{Sty}_{\mathsf{cl},\pi} \wedge \bigwedge_\mathsf{k} \mathbf{Sty}_{\pi_\mathsf{k},\mathsf{cl}_\mathsf{k}} \wedge \mathbf{FinalIn}_{\mathbf{name}(\mathsf{cld}),\mathsf{F}} \mid \exists \mathsf{cld} \in \mathbf{clds}(\mathsf{F}), \\
 & \qquad \exists \mathsf{cl}, \overline{\mathsf{cl}_\mathsf{k}}^\mathsf{k}, \overline{\mathsf{v}_\mathsf{k}}^\mathsf{k} \mathsf{cl\ m}(\overline{\mathsf{cl}_\mathsf{k} \mathsf{v}_\mathsf{k}}^\mathsf{k}) \in \mathbf{mds}(\mathsf{cld})\} \vee \\
 & \quad \bigvee\{\mathbf{Sty}_{\mathsf{cl},\pi} \wedge \bigwedge_\mathsf{k} \mathbf{Sty}_{\pi_\mathsf{k},\mathsf{cl}_\mathsf{k}} \wedge \mathbf{Final}_{\mathbf{name}(\mathsf{rcld}),\mathsf{F}} \mid \exists \mathsf{rcld} \in \mathbf{rclds}(\mathsf{F}), \\
 & \qquad \exists \mathsf{cl}, \overline{\mathsf{cl}_\mathsf{k}}^\mathsf{k}, \overline{\mathsf{v}_\mathsf{k}}^\mathsf{k} \mathsf{cl\ m}(\overline{\mathsf{cl}_\mathsf{k} \mathsf{v}_\mathsf{k}}^\mathsf{k}) \in \mathbf{mds}(\mathsf{rcld})\} \\
\mathbf{defined}(\mathsf{cl}) & \Rightarrow \bigvee\{\mathbf{In}_\mathsf{F} \mid \exists \mathsf{cld} \in \mathbf{clds}(\mathsf{F}), \mathbf{name}(\mathsf{cld}) = \mathsf{cl}\} \\
\tau\ \mathbf{introduces}\ \mathsf{ms}\ \mathbf{before}\ \mathsf{F} & \Rightarrow \bigvee\{\mathbf{In}_\mathsf{G} \wedge \mathbf{Prec}_{\mathsf{G},\mathsf{F}} \wedge \bigwedge\{\mathbf{In}_\mathsf{H} \to \mathbf{Prec}_{\mathsf{F},\mathsf{H}} \vee \mathbf{Prec}_{\mathsf{H},\mathsf{G}} \mid \exists \mathsf{cld}' \in \mathbf{clds}(\mathsf{H}), \mathbf{type}(\mathbf{name}(\mathsf{cld}')) = \tau\} \\
 & \qquad \mid \exists \mathsf{cld} \in \mathbf{clds}(\mathsf{G}), \mathbf{type}(\mathbf{name}(\mathsf{cld})) = \tau \wedge \mathsf{ms} \in \mathbf{methods}(\mathbf{mds}(\mathsf{cld}))\} \vee \\
 & \quad \bigvee\{\mathbf{In}_\mathsf{G} \wedge \mathbf{Prec}_{\mathsf{G},\mathsf{F}} \wedge \bigwedge\{\mathbf{In}_\mathsf{H} \to \mathbf{Prec}_{\mathsf{F},\mathsf{H}} \vee \mathbf{Prec}_\mathsf{H}, \mathsf{G} \mid \exists \mathsf{cld}' \in \mathbf{clds}(\mathsf{H}), \mathbf{type}(\mathbf{name}(\mathsf{cld}')) = \tau\} \\
 & \qquad \mid \exists \mathsf{rcld} \in \mathbf{rclds}(\mathsf{G}), \mathbf{type}(\mathbf{name}(\mathsf{rcld})) = \tau \wedge \mathsf{ms} \in \mathbf{methods}(\mathbf{mds}(\mathsf{rcld}))\} \\
\mathsf{dcl}\ \mathbf{introduced\ before}\ \mathsf{F} & \Rightarrow \bigvee\{\mathbf{In}_\mathsf{G} \wedge \mathbf{Prec}_{\mathsf{G},\mathsf{F}} \mid \exists \mathsf{cld} \in \mathbf{clds}(\mathsf{F}), \mathbf{name}(\mathsf{cld}) = \mathsf{dcl}\} \\
\mathsf{cl\ f}\ \mathbf{unique\ in}\ \mathsf{dcl} & \Rightarrow \bigwedge\{\neg \mathbf{In}_\mathsf{F} \mid \exists \mathsf{cld} \in \mathbf{clds}(\mathsf{F}), \mathbf{name}(\mathsf{cld}) = \mathsf{dcl} \wedge \exists \mathsf{cl}', \mathsf{cl}'\mathsf{f} \in \mathbf{fds}(\mathsf{cld}) \wedge \mathsf{cl} \neq \mathsf{cl}'\} \wedge \\
 & \quad \bigwedge\{\neg \mathbf{In}_\mathsf{F} \mid \exists \mathsf{rcld} \in \mathbf{rclds}(\mathsf{F}), \mathbf{name}(\mathsf{rcld}) = \mathsf{dcl} \wedge \exists \mathsf{cl}', \mathsf{cl}'\mathsf{f} \in \mathbf{fds}(\mathsf{rcld}) \wedge \mathsf{cl} \neq \mathsf{cl}'\} \\
\mathsf{cl\ m}\ (\overline{\mathsf{vd}_\mathsf{k}}^\mathsf{k})\ \mathbf{unique\ in}\ \mathsf{dcl} & \Rightarrow \bigwedge\{\neg \mathbf{In}_\mathsf{F} \mid \exists \mathsf{cld} \in \mathbf{clds}(\mathsf{F}), \mathbf{name}(\mathsf{cld}) = \mathsf{dcl} \wedge \exists \mathsf{cl}', \overline{\mathsf{vd}'_\mathsf{k}}^\mathsf{k}, \mathsf{cl}'\mathsf{m}\ (\overline{\mathsf{vd}'_\mathsf{k}}^\mathsf{k}) \in \mathbf{mds}(\mathsf{cld}) \wedge \mathsf{cl} \neq \mathsf{cl}' \vee \\
 & \qquad (\bigvee_\mathsf{k} \mathsf{vd}_\mathsf{k} \neq \mathsf{vd}'_\mathsf{k})\} \wedge \\
 & \quad \bigwedge\{\neg \mathbf{In}_\mathsf{F} | \exists \mathsf{rcld} \in \mathbf{rclds}(\mathsf{F}), \mathbf{name}(\mathsf{rcld}) = \mathsf{dcl} \wedge \exists \mathsf{cl}', \overline{\mathsf{vd}'_\mathsf{k}}^\mathsf{k}, \mathsf{cl}'\mathsf{m}\ (\overline{\mathsf{vd}'_\mathsf{k}}^\mathsf{k}) \in \mathbf{mds}(\mathsf{rcld}) \wedge \mathsf{cl} \neq \mathsf{cl}' \vee \\
 & \qquad (\bigvee_\mathsf{k} \mathsf{vd}_\mathsf{k} \neq \mathsf{vd}'_\mathsf{k})\} \\
\mathsf{f} \notin \mathbf{fields}(\mathbf{parent}(\mathsf{dcl})) & \Rightarrow \bigwedge\{\mathbf{In}_\mathsf{F} \wedge \mathbf{FinalIn}_{\mathbf{name}(\mathsf{cld}),\mathsf{F}} \to \neg \mathbf{Sty}_{\mathbf{type}(\mathsf{dcl}),\mathsf{cl}} \mid \\
 & \qquad \exists \mathsf{cld} \in \mathbf{clds}(\mathsf{F}), \mathbf{name}(\mathsf{cld}) = \mathsf{cl} \wedge \mathsf{dcl} \neq \mathsf{cl} \wedge \exists \mathsf{cl}', \mathsf{cl}'\mathsf{f} \in \mathbf{fds}(\mathsf{cld})\} \wedge \\
 & \quad \bigwedge\{\mathbf{In}_\mathsf{F} \wedge \mathbf{Final}_{\mathbf{name}(\mathsf{rcld}),\mathsf{F}} \to \neg \mathbf{Sty}_{\mathbf{type}(\mathsf{dcl}),\mathsf{cl}} \mid \\
 & \qquad \exists \mathsf{rcld} \in \mathbf{rclds}(\mathsf{F}), \mathbf{name}(\mathsf{rcld}) = \mathsf{cl} \wedge \mathsf{dcl} \neq \mathsf{cl} \wedge \exists \mathsf{cl}', \mathsf{cl}'\mathsf{f} \in \mathbf{fds}(\mathsf{rcld})\} \\
\mathbf{pmtype}(\mathsf{dcl}, \mathsf{m}) = \tau & \Rightarrow \bigwedge\{\mathbf{In}_\mathsf{F} \wedge \mathbf{FinalIn}_{\mathbf{name}(\mathsf{cld}),\mathsf{F}} \to \neg \mathbf{Sty}_{\mathbf{type}(\mathsf{dcl}),\mathsf{cl}} \mid \exists \mathsf{cld} \in \mathbf{clds}(\mathsf{F}), \mathbf{name}(\mathsf{cld}) = \mathsf{cl} \\
 & \qquad \wedge \mathsf{dcl} \neq \mathsf{cl} \wedge \mathsf{m} \in \mathbf{methods}(\mathsf{cld}) \wedge \mathbf{mtype}(\mathsf{cld}, \mathsf{m}) \neq \tau \\
 & \quad \bigwedge\{\mathbf{In}_\mathsf{F} \wedge \mathbf{Final}_{\mathbf{name}(\mathsf{cld}),\mathsf{F}} \to \neg \mathbf{Sty}_{\mathbf{type}(\mathsf{dcl}),\mathsf{cl}} \mid \exists \mathsf{rcld} \in \mathbf{rclds}(\mathsf{F}), \mathbf{name}(\mathsf{rcld}) = \mathsf{cl} \\
 & \qquad \wedge \mathsf{dcl} \neq \mathsf{cl} \wedge \mathsf{m} \in \mathbf{methods}(\mathsf{rcld}) \wedge \mathbf{mtype}(\mathsf{rcld}, \mathsf{m}) \neq \tau
\end{array}
$$

where

$$
\begin{array}{ll}
\mathbf{FinalIn}_{\mathsf{cl},\mathsf{F}} & \leftrightarrow \mathbf{In}_\mathsf{F} \wedge \bigwedge\{\mathbf{In}_\mathsf{G} \to \mathbf{Prec}_{\mathsf{G},\mathsf{F}} \mid \mathsf{cl} \in \mathbf{names}(\mathbf{clds}(\mathsf{G})) \wedge \mathsf{G} \neq \mathsf{F}\} \\
\mathbf{Final}_{\mathsf{cl},\mathsf{F}} & \leftrightarrow \mathbf{In}_\mathsf{F} \wedge \bigwedge\{\mathbf{In}_\mathsf{G} \to \mathbf{Prec}_{\mathsf{G},\mathsf{F}} \mid \mathsf{cl} \in \mathbf{names}(\mathbf{clds}(\mathsf{G}))\}
\end{array}
$$

Figure 6.6: Translation of constraints to propositional formulas.

closure of the subtyping relation, starting with the parent/child relationships established by the last definition of a class in a product specification. A satisfying assignment to $\mathsf{WF}_{\mathsf{Spec}}$, the conjunction of all these constraints, represents a unique valid product specification.

Checking that the product line is contained in the set of type-safe programs thus reduces to checking the validity of $\mathsf{FM} \wedge \mathsf{WF}_{\mathsf{Spec}} \to \bigwedge_\mathsf{F} \mathbf{In}_\mathsf{F} \to \mathbf{Propify}(\mathcal{C}_\mathsf{F})$. The left side of the implication restricts truth assignments to valid product specifications of the feature model FM, while the right side ensures that the product specification is in the set of type-safe programs. A falsifying assignment corresponds to a member of the product line which isn't type-safe; this assignment can be used to determine the exact source of the typing

$$\begin{array}{rl}
\text{PREC\_TOTAL:} & \forall \mathsf{A}, \mathsf{B}, \mathsf{A} \neq \mathsf{B}, \mathbf{In}_{\mathsf{A}} \land \mathbf{In}_{\mathsf{B}} \leftrightarrow (\mathbf{Prec}_{\mathsf{A},\mathsf{B}} \lor \mathbf{Prec}_{\mathsf{B},\mathsf{A}}) \\
\text{PREC\_ASYM:} & \forall \mathsf{A}, \mathsf{B}, \mathbf{Prec}_{\mathsf{A},\mathsf{B}} \rightarrow \neg \mathbf{Prec}_{\mathsf{B},\mathsf{A}} \\
\text{PREC\_IRREFL:} & \forall \mathsf{A}, \neg \mathbf{Prec}_{\mathsf{A},\mathsf{A}} \\
\text{STY\_REFL:} & \forall \tau, \mathbf{Sty}_{\tau,\tau} \leftrightarrow \bigvee\{\mathbf{In}_{\mathsf{F}} \mid \mathsf{cld} \in \mathbf{clds}(\mathsf{F}) \land \mathbf{type}(\mathbf{name}(\mathsf{cld})) = \tau\} \\
\text{STY\_OBJ:} & \mathbf{Sty}_{\mathsf{Object},\mathsf{Object}} \\
\text{STY\_ASYM:} & \forall \tau_1, \tau_2, \mathbf{Sty}_{\tau_1,\tau_2} \rightarrow \neg \mathbf{Sty}_{\tau_2,\tau_1} \\
\text{STY\_TOTAL:} & \forall \tau_1, \tau_2, \tau_3, \mathbf{Sty}_{\tau_1,\tau_2} \leftrightarrow ((\mathbf{Sty}_{\tau_1,\tau_3} \land \mathbf{Sty}_{\tau_3,\tau_2}) \lor \\
 & \bigvee\{\mathbf{In}_{\mathsf{F}} \mid \exists \mathsf{cld} \in \mathbf{clds}(\mathsf{F}), \mathbf{type}(\mathbf{name}(\mathsf{cld})) = \tau_1 \land \mathbf{type}(\mathbf{parent}(\mathsf{cld})) = \tau_2\} \land \\
 & \bigwedge\{\mathbf{In}_{\mathsf{G}} \rightarrow \mathbf{Prec}_{\mathsf{G},\mathsf{F}} \mid \mathsf{G} \neq \mathsf{F} \land \exists \mathsf{cld} \in \mathbf{clds}(\mathsf{G}), \mathbf{type}(\mathbf{name}(\mathsf{cld})) = \tau_1\} \land \\
 & \bigwedge\{\mathbf{In}_{\mathsf{G}} \rightarrow \mathbf{Prec}_{\mathsf{G},\mathsf{F}} \mid \mathsf{G} \neq \mathsf{F} \land \exists \mathsf{rcld} \in \mathbf{rclds}(\mathsf{G}), \mathbf{type}(\mathbf{name}(\mathsf{rcld})) = \tau_1\} \lor \\
 & \bigvee\{\mathbf{In}_{\mathsf{F}} \mid \exists \mathsf{rcld} \in \mathbf{rclds}(\mathsf{F}), \mathbf{type}(\mathbf{name}(\mathsf{rcld})) = \tau_1 \land \mathbf{type}(\mathbf{parent}(\mathsf{cld})) = \tau_2 \land \\
 & \mathbf{name}(\mathsf{rcld}) \notin \mathbf{names}(\mathbf{clds}(\mathsf{F}))\} \land \\
 & \bigwedge\{\mathbf{In}_{\mathsf{G}} \rightarrow \mathbf{Prec}_{\mathsf{G},\mathsf{F}} \mid \mathsf{G} \neq \mathsf{F} \land \exists \mathsf{cld} \in \mathbf{clds}(\mathsf{G}), \mathbf{type}(\mathbf{name}(\mathsf{cld})) = \tau_1\} \land \\
 & \bigwedge\{\mathbf{In}_{\mathsf{G}} \rightarrow \mathbf{Prec}_{\mathsf{G},\mathsf{F}} \mid \mathsf{G} \neq \mathsf{F} \land \exists \mathsf{rcld} \in \mathbf{rclds}(\mathsf{G}), \mathbf{type}(\mathbf{name}(\mathsf{rcld})) = \tau_1\}\ ) \\
\text{STY\_WF:} & \forall \mathsf{A}, \forall \mathsf{c} \in \mathbf{clds}(\mathsf{A}), \mathbf{In}_{\mathsf{A}} \rightarrow \mathbf{Sty}_{\mathbf{ty}(\mathbf{name}(\mathsf{c})),\mathsf{Object}}
\end{array}$$

Figure 6.7: Constraints on the precedence and subtyping relations.

problem.

While checking the validity of this formula is co-NP-complete, the SAT instances generated by our approach are highly structured. This makes them amenable to fast analysis by modern SAT solvers, as demonstrated by an implementation of a system based on this approach for checking safe composition of AHEAD software product lines [TBKC07]. The size statistics for the four product lines analyzed are presented in Table 6.1. The tools identified several errors in the existing feature models of these product lines. It took less than 30 seconds to analyze the code, generate the SAT formula, and run the SAT solver for JPL, the largest product line. This is less than the time it took to generate and compile a single program in the product line.

Product Line	# of Features	# of Prog.	Code Base Jak/Java LOC	Program Jak/Java LOC
PPL	7	20	2000/2000	1K/1K
BPL	17	8	12K/16K	8K/12K
GPL	18	80	1800/1800	700/700
JPL	70	56	34K/48K	22K/35K

Table 6.1: Product Line Statistics from [12].

A similar reduction applies to the feature module system presented earlier. The reduction to SAT relies on first capturing variability in module interfaces [KOE12a]. Under this model, the $\mathsf{imports}$ and $\mathsf{exports}$ of a single module depend on a selection of configuration options. Figure 6.8 presents the machinery needed to add variability to semantic and syntactic feature modules. The key additions are a set of configuration options, $\mathcal{F}$, which govern what is imported/assumed and exported/proved by a module. The $\mathsf{configure}$ function generates a module in the original module system given a set of configurations.

A module with $|\mathcal{S}|$ configuration options can be configured to build $2^{|\mathcal{S}|}$ possible modules. For two sets of features $\mathcal{S}$ and $\mathcal{T}$, we can capture the composition of all combinations of all subsets of $\mathcal{S}$ and $\mathcal{T}$ by representing each as a module with interface variability, $\mathsf{m}_{\mathcal{S}}$ and $\mathsf{m}_{\mathcal{T}}$ and composing the two $\mathsf{m}_{\mathcal{S}} + \mathsf{m}_{\mathcal{T}}$, reducing the number of interface checks from $2^{|\mathcal{S}|} + 2^{|\mathcal{T}|}$ to 1. Variable modules representing all the feature modules in a set $\mathcal{S}$ can be built by choosing a distinguished configuration option for each feature module and adding it to the exports and imports of each feature module. A single variable module can then be built representing the entire $2^{|\mathcal{S}|}$ possible compositions by combining the $|\mathcal{S}|$ feature modules together. To check safe composition of a module with variability, it suffices to check that all configuration options allowed by the feature model have produced semantic modules with empty assume interface and can thus export a proof of the property of interest. This can be done by checking the satisfiability of $\mathsf{FM} \wedge \bigwedge_{\mathsf{F} \in \overline{\mathsf{F}}} \mathsf{F}$ using a SAT solver. While in the worst case, this remains exponential in $|\mathcal{S}|$ SAT solvers have been shown to be efficient at dealing with product-line combinatorics in practice [Jan10, ARW+13].

$$\begin{array}{ll}
\mathsf{F}, \mathsf{G}, \mathsf{H} \in \mathcal{F} & \text{Configuration Options} \\
\mathsf{e} \triangleq \mathsf{x} \mapsto \mathsf{f} & \text{Syntactic Exports} \\
\mu_\delta = \{\overline{\mathsf{F} \rightarrow \mathsf{v}}, \overline{\mathsf{G} \rightarrow \mathsf{e}}\} \in \mathcal{M}_\delta & \text{Variable Syntactic Modules}
\end{array}$$

$$\begin{array}{ll}
\mathsf{configure}(\mu_\delta, \overline{\mathsf{H}}) \triangleq \{\{\mathsf{v} \mid \mathsf{F} \in \mathsf{H}\}, \{\mathsf{e} \mid \mathsf{G} \in \mathsf{H}\}\} & \text{Module Configuration} \\
\mathsf{imports}(\mu_\delta, \overline{\mathsf{H}}) \triangleq \mathsf{imports}(\mathsf{configure}(\mu_\delta, \overline{\mathsf{H}})) & \text{Configured Module Imports} \\
\mathsf{exports}(\mu_\delta, \overline{\mathsf{H}}) \triangleq \mathsf{exports}(\mathsf{configure}(\mu_\delta, \overline{\mathsf{H}})) & \text{Configured Module Exports}
\end{array}$$

$$\frac{\forall\ \overline{\mathsf{H}}, \mathsf{configure}(\mu_\delta, \overline{\mathsf{H}})\ \mathsf{OK}}{\mu_\delta\ \mathsf{OK}}$$

$$\frac{\begin{array}{c}
\bar{\mathsf{I}} = \overline{\mathsf{F_m} \bigwedge_{\mathsf{G} \rightarrow \mathsf{v_m} \mapsto \mathsf{f_n} \in \overline{\mathsf{E}}} \neg \mathsf{G} \rightarrow \mathsf{v_m}} \cup \overline{\mathsf{F_n} \bigwedge_{\mathsf{G} \rightarrow \mathsf{v_n} \mapsto \mathsf{f_n} \in \overline{\mathsf{E}}} \neg \mathsf{G} \rightarrow \mathsf{v_n}} \\
\overline{\mathsf{E}} = \{\mathsf{G_m} \rightarrow \mathsf{e_m} | \mathsf{e_m} \notin \overline{\mathsf{e_n}}\} \cup \{\mathsf{G_n} \rightarrow \mathsf{e_n} | \mathsf{e_n} \notin \overline{\mathsf{e_m}}\} \cup \\
\{(\mathsf{G_m} \wedge \neg \mathsf{G_n}) \vee (\mathsf{G_n} \wedge \neg \mathsf{G_m}) \rightarrow \mathsf{e} | \mathsf{e} \in \mathsf{e_m} \wedge \mathsf{e_n}\} \\
\forall \overline{\mathsf{H}}, \mathsf{acyclic} \begin{pmatrix} \mathsf{configure}(\{\overline{\mathsf{F_m} \rightarrow \mathsf{v_m}}, \overline{\mathsf{G_m} \rightarrow \mathsf{e_m}}\}, \overline{\mathsf{H}}), \\ \mathsf{configure}(\{\overline{\mathsf{F_n} \rightarrow \mathsf{v_n}}, \overline{\mathsf{G_n} \rightarrow \mathsf{e_n}}\}, \overline{\mathsf{H}}) \end{pmatrix}
\end{array}}{\{\overline{\mathsf{F_m} \rightarrow \mathsf{v_m}}, \overline{\mathsf{G_m} \rightarrow \mathsf{e_m}}\} + \{\overline{\mathsf{F_n} \rightarrow \mathsf{v_n}}, \overline{\mathsf{G_n} \rightarrow \mathsf{e_n}}\} = \{\bar{\mathsf{I}},\ \overline{\mathsf{E}}\}}$$

$$\begin{array}{ll}
\mu_\pi = \{\mu_\delta, \overline{\mathsf{F} \rightarrow \mathsf{u}\ :\ \pi_\mathsf{v}}, \overline{\mathsf{G} \rightarrow \mathsf{w} \mapsto \rho\ :\ \pi_\mathsf{x}}\} & \text{Variable Semantic Modules}
\end{array}$$

$$\begin{array}{ll}
\mathsf{configure}(\mu_\pi, \overline{\mathsf{H}}) \triangleq & \text{Module Configuration} \\
\quad \{\mathsf{configure}(\mu_\delta), \{\mathsf{u}\ :\ \pi_\mathsf{v} \mid \mathsf{F} \in \mathsf{H}\}, \{\mathsf{w} \mapsto \rho\ :\ \pi_\mathsf{x} \mid \mathsf{G} \in \mathsf{H}\}\} & \\
\mathsf{assumes}(\mu_\pi, \overline{H}) \triangleq \mathsf{assumes}(\mathsf{configure}(\mu_\pi, \overline{H})) & \text{Configured Module Imports} \\
\mathsf{proves}(\mu_\pi, \overline{H}) \triangleq \mathsf{proves}(\mathsf{configure}(\mu_\pi, \overline{H})) & \text{Configured Module Exports}
\end{array}$$

$$\frac{\forall\ \overline{\mathsf{H}}, \mathsf{configure}(\mu_\pi, \overline{\mathsf{H}})\ \mathsf{OK}}{\mu_\pi\ \mathsf{OK}}$$

$$\frac{\begin{array}{c}
\bar{\mathsf{I}} = \overline{\mathsf{F_m} \bigwedge_{\mathsf{G} \rightarrow \mathsf{u_m}\ :\ \pi_\mathsf{vm} \in \overline{\mathsf{E}}} \neg \mathsf{G} \rightarrow \mathsf{u_m}\ :\ \pi_\mathsf{mv}} \cup \overline{\mathsf{F_n} \bigwedge_{\mathsf{G} \rightarrow \mathsf{u_n}\ :\ \pi_\mathsf{vn} \in \overline{\mathsf{E}}} \neg \mathsf{G} \rightarrow \mathsf{u_n}\ :\ \pi_\mathsf{nv}} \\
\overline{\mathsf{E}} = \{\mathsf{G_m} \rightarrow \mathsf{w_m} \mapsto \rho_\mathsf{m}\ :\ \pi_\mathsf{mx} | \mathsf{w_m} \notin \overline{\mathsf{w_n}}\} \cup \{\mathsf{G_n} \rightarrow \mathsf{w_n} \mapsto \rho_\mathsf{n}\ :\ \pi_\mathsf{nx} | \mathsf{w_n} \notin \overline{\mathsf{w_m}}\} \cup \\
\{(\mathsf{G_m} \wedge \neg \mathsf{G_n}) \vee (\mathsf{G_n} \wedge \neg \mathsf{G_m}) \rightarrow \mathsf{w} \mapsto \rho\ :\ \pi_\mathsf{x} | \mathsf{w} \in \overline{\mathsf{w_m}} \wedge \mathsf{w} \in \overline{\mathsf{w_n}}\} \\
\forall \overline{\mathsf{H}}, \mathsf{consistent} \begin{pmatrix} \mathsf{configure}(\{\overline{\mathsf{F_m} \rightarrow \mathsf{u_m}\ :\ \pi_\mathsf{mv}}, \overline{\mathsf{G_m} \rightarrow \mathsf{w_m} \mapsto \rho_\mathsf{m}\ :\ \pi_\mathsf{mx}}\}, \overline{\mathsf{H}}), \\ \mathsf{configure}(\{\overline{\mathsf{F_n} \rightarrow \mathsf{u_n}\ :\ \pi_\mathsf{nv}}, \overline{\mathsf{G_n} \rightarrow \mathsf{w_n} \mapsto \rho_\mathsf{n}\ :\ \pi_\mathsf{nx}}\}, \overline{\mathsf{H}}) \end{pmatrix}
\end{array}}{\begin{array}{c}
\{\mu_\delta, \overline{\mathsf{F_m} \rightarrow \mathsf{u_m}\ :\ \pi_\mathsf{mv}}, \overline{\mathsf{G_m} \rightarrow \mathsf{w_m} \mapsto \rho_\mathsf{m}\ :\ \pi_\mathsf{mx}}\} + \\
\{\nu_\delta, \overline{\mathsf{F_n} \rightarrow \mathsf{u_n}\ :\ \pi_\mathsf{nv}}, \overline{\mathsf{G_n} \rightarrow \mathsf{w_n} \mapsto \rho_\mathsf{n}\ :\ \pi_\mathsf{nx}}\} = \\
\{\mu_\delta + \nu_\delta, \bar{\mathsf{I}},\ \overline{\mathsf{E}}\}
\end{array}}$$

Figure 6.8: Definitions for Feature Modules with Variability

Chapter 7

Related Work

This dissertation lies at the intersection of programming languages, theorem proving, and software engineering, resulting in a large body of related work.

7.1 Feature Oriented Design

In Feature Oriented Design (FOD), systems are specified as a selection of features, typically as expressions in the Feature Interaction Algebra [BHMZ13, BHK11]. Concrete systems are synthesized from these specifications, akin to how query plans are synthesized from queries expressed in relational algebra. Approaches to synthesis from these specifications can be divided into two main groups: *projectional* and *compositional*.

Projectional Approaches In projectional approaches all possible feature variations are expressed using a meta-language to annotate (or color) a single system with variability. A preprocessor then projects out a concrete system based on a feature specification. C-style `#ifdef` statements are a common way to implement projectional systems. Because systems with variability are expressed in a meta-language, they do not have access to analyses (i.e. type checkers) for the target domain. While it is easy to analyze a given configuration, developing efficient analyses for entire families remains an open research area. There have been a number of techniques proposed for parsing [KGR+11] and typechecking [KOE12b] C code with `#ifdef` variability. Kastner et. al have developed a type system which preserves typing of projections for Colored Featherweight Java [AKL08], a Featherweight Java variant augmented with coloring annotations.

Compositional Approaches In compositional approaches, synthesis is realized as a homomorphic mapping of features to modules in the target domain. These modules are then composed together to build the desired product. The AHEAD system [Bat04] implements feature modules of a target domain using a distinct domain specific language, each of which have their own composition functions. FeatureHouse [AKL09] is a tool for generically specifying feature modules and composition. It has a generic representation of hierarchically structured documents called *feature structure trees* (FSTs), which are variants of ASTs; these trees are composed together using a superimposition operator. After showing how to map a desired syntactic domain onto FSTs, users get this composition operator "for free". FeatureHouse and AHEAD are both strictly syntactic- there is no semantic information attached to FSTs or AHEAD's DSLs outside of structural properties. The purely syntactic implementation of module composition creates an impediment to product-line analyses.

A couple of module systems which enable true semantic modularity for software have been proposed. *g*DEEP [AH07] is a language-independent calculus designed to capture the core ideas of feature refinement. The type system for *g*DEEP transfers information across feature boundaries and is combined with the type system for an underlying language to type feature compositions.

The module system proposed by Anacona et. al to type check, compile, and link source code fragments [AD05] hews closely to the LFJ. Their source code fragments could reference external class definitions, requiring other fragments to be included in order to build a well-typed program. Code fragments are compiled into bytecode fragments augmented with typing constraints that ranged over type variables, similar to the constraints used in the LFJ typing rules. Linking of individually compiled bytecode fragments corresponds to module composition, with constraints being resolved during. If all the constraints are resolved during linking, the resulting code is the same as if all the pieces had been globally compiled.

7.2 Modular Semantics

Modular programming language semantics first came into focus in the context of compiler construction. The traditional domains of denotational semantics, Scott domains [Sto77], were ill-suited for generating efficient compilers [Lee89] because, among other reasons, the domain must be rewritten if the model is insufficient to express effects. This lead to the development of a hybrid approach known as action semantics [Mos92], which include a macrosemantics for describing a high-level meaning using operations called *facets*, and microsemantics for describing the meaning of those operations. Separating operations from their precise meaning made it possible to add new operations to the macrosemantic domain.

Mosses' Modular Structural Operational Semantics (MSOS) [Mos04] are an evolution of action semantics designed to facilitate the modular development of programming language semantics. In this paradigm, rules are written with an abstract label which effectively serves as a repository for all effects, allowing rules to be written once and reused with different instantiations depending on the effects supported by the final language. Effect-free transitions pass around the labels of their subexpressions:

$$\frac{d \xrightarrow{X} d'}{\texttt{let}\ d\ \texttt{in}\ e \xrightarrow{X} \texttt{let}\ d'\ \texttt{in}\ e} \qquad \text{(R-LetB)}$$

Those rules which rely on an effectual transition specify that the final labeling supports effects:

$$\frac{e \xrightarrow{\{p=p_1[p_0]\ldots\}} e'}{\texttt{let}\ p_0\ \texttt{in}\ e \xrightarrow{\{p=p_1\ldots\}} \texttt{let}\ p_0\ \texttt{in}\ e} \qquad \text{(R-LetE)}$$

While action semantics and MSOS are well-suited to modular language specification, their use in metatheory remains an open question, as Mosses notes [Mos92]: "Although the foundations of action semantics are firm enough, the theory for reasoning about actions (and hence about programs) is still rather weak, and needs further development."

Stärk et. al [SSB01] develop a complete Java 1.0 compiler through incremental refinement of a set of Abstract State Machines. Starting with `ExpI`, a core language of imperative Java expressions which contains a grammar, interpreter, and compiler, the authors add features which incrementally update the language until an interpreter and compiler are derived for the full Java 1.0 specification. The authors then write a monolithic proof of correctness for the full language. Later work casts this approach in the calculus of features [BB08], noting that the pen-and-paper proof could have been developed incrementally as well.

Monads Monads are an alternate approach for modeling computational effects first proposed by Moggi [Mog89] and popularized by Wadler [Wad92b]. Various researchers (e.g., [JD93, Ste94]) have sought to modularize monads in order to construct modular semantic domains. Monad transformers emerged [CM93, LHJ95] from this process, and in later years various alternative implementation designs facilitating monad (transformer) implementations, have been developed, including Filinksi's layered monads [Fil99] and Jaskelioff's Monatron [Jas11]. Both Schrijvers and Oliveira [SO10] and later Bahr and Hvitved [BH11] have shown how to define modular semantics with monads for effects; this is essentially the approach used here for abstracting semantic domains over effects.

Monads and Subtyping Filinski's MultiMonadic MetaLanguage (M^3L) [Fil07, Fil10] embraces the monadic approach, but uses subtyping (or subeffecting) to combine the effects of different components. The subtyping relation is fixed at the program or language level, which does not provide the adaptability we achieve with constrained polymorphism.

Algebraic Effects and Effect Handlers In the semantics community the algebraic theory of computational effects [PP02] has been an active area of research. Many of the laws about effects, which we have not seen before in the context of functional programming, can be found throughout the semantics literature. Our first four laws for exceptions, for example, have been presented by Levy [Lev06].

A more recent model of side effects are effect handlers. They were introduced by Plotkin and Pretnar [PP09] as a generalization from exception handlers to handlers for a range of computational effects, such as I/O, state, and nondeterminism. Bauer and Pretnar [BP12] built the language *Eff* around effect handlers and show how to implement a wide range of effects in it. Kammar et. al [KLO12] showed that effect handlers can be implemented in terms of delimited continuations or free monads. The major advantage of effect handlers over monads is that they are more easily composed, as any composition of effect operations and corresponding handlers is valid. In contrast, not every composition of monads is a monad. Effect handlers could potentially reduce the amount of work involved on proofs about interactions of effects.

Other Effect Models Other useful models have been proposed, such as *applicative functors* [MP08] and *arrows* [Hug00], each with their own axioms and modularity properties.

7.2.1 Effects and Reasoning

Non-Modular Monadic Reasoning Although monads are a purely functional way to encapsulate computational-effects, programs using monads are challenging to reason about. The main issue is that monads provide an abstraction over purely functional models of effects, allowing functional programmers to write programs in terms of abstract operations like |»=|, |return|, or |get| and |put|. One way to reason about monadic programs is to remove the abstraction provided by such operations [HF08]. However, this approach is fundamentally non-modular.

Modular Monadic Reasoning One approach to modular monadic reasoning is to exploit *parametricity* [Rey83, Wad89]. Voigtländer [Voi09] has shown how to derive parametricity theorems for type constructor classes such as |Monad|. Unfortunately, the reasoning power of parametricity is limited, and parametricity is not supported by proof assistants like Coq.

A second technique uses *algebraic laws*. Liang and Hudak [LH96] present one of the earliest examples of using algebraic laws for reasoning. They use algebraic laws for reader monads to prove correctness properties about a modular compiler. In contrast to our work, their compiler correctness proofs are pen-and-paper and thus more informal than our proofs. Since they are not restricted by a termination checker or the use of positive types only, they exploit features like general recursion in their definitions. Oliveira et. al [OSC10] have also used algebraic laws for the state monad, in combination with parametricity, for modular proofs of non-interference of aspect-oriented advice. Hinze and Gibbons discuss several other algebraic laws for various types of monads [GH11].

Formalization of Monad Transformers Huffmann [Huf12] illustrates an approach for mechanizing type constructor classes in Isabelle/HOL with monad transformers. He considers transformer variants of the resumption, error and writer monads, but features only the generic functor, monad and transformer laws. The work tackles many issues that are not relevant for our Coq setting, such as lack of parametric polymorphism and explicit modeling of laziness.

7.3 Modularity in Mechanized Semantics

Several ad-hoc tool-based approaches provide reuse, but none is based on a proof assistant's modularity features alone. The Tinkertype project [LP03] is a framework for modularly specifying formal languages. It was used to format the language variants used in Pierce's "Types and Programming Languages" [Pie02], and to compose traditional pen-and-paper proofs. The Ott tool [S$^+$07] allows users to write definitions and theorem statements in an ASCII format designed to mirror pen-and-paper formalizations. These are then automatically translated to definitions in either LaTeX or a theorem prover, and proofs and functions are then written using the generated definitions.

Both Boite [Boi04] and Mulhern [Mul06] consider how to extend existing inductive definitions and reuse related proofs in the Coq proof assistant. Both only consider adding new cases and rely on the critical observation that proofs over the extended language can be patched by adding pieces for the new cases. The latter promotes the idea of "proof weaving" for merging inductive definitions of two languages which merges proofs from each by case splitting and reusing existing proof terms. An unimplemented tool is proposed to automatically weave definitions together. The former extends Coq with a new `Extend` keyword that redefines an existing inductive type with new cases and a `Reuse` keyword that creates a partial proof for an extended datatype with proof holes for the new cases which the user must interactively fill in. These two keywords explicitly extend a concrete definition and thus modules which use them cannot be checked by Coq independently of those definitions. This presents a problem when building a language product line: adding a new feature to a base language can easily break the proofs of subsequent features which are written using the original, fixed language. Interactions can also require updates to existing features in order to layer them onto the feature enhanced base language, leading to the development of parallel features that are applied depending on whether the new feature is included. These keyword extensions were written for a previous version of Coq and are not available for the current version of the theorem prover.

Chlipala [Chl10] proposes a using adaptive tactics written in Coq's tactic definition language LTac [Del00] to achieve proof reuse for a certified compiler. The generality of the approach is tested by enhancing the original language with let expressions, constants, equality testing, and recursive functions, each of which required relatively minor updates to existing proof scripts. In contrast to our modular datatype approach, each refinement was incorporated into a new monolithic language, with the new variant having a distinct set of proofs to maintain. Our case study uses adaptive proofs hooked into Coq's type class mechanism to automatically dispatch some proof algebra obligations, in particular algebras for inversion lemmas.

Both Schwaab et. al [SS12] and Keuchel [KS13] developed alternate approaches to modularizing metatheory in Agda based on encoding datatypes using universes in contrast to the church-encoded datatypes presented here. This approach has the benefit of avoiding universal properties, although the extra layer of abstraction can make reasoning over universe-encoded datatypes cumbersome.

Transparency One long-standing criticism of mechanized metatheory has been that it interferes with adequacy, i.e. convincing users that the proven theorem is in fact the desired one [Pol98]. Modular inductive datatypes have the potential for exacerbating transparency concerns, as the encodings are distributed over different components. Combining a higher-level notation provided by a tool like Ott with our semantic composition mechanisms could be interesting direction for future work. Such a higher-level notation could help with transparency; while proper composition mechanisms could help with generating modular code for Ott specifications.

7.4 Product Line Analyses

Representing feature models as propositional formulas in order to verify their consistency was first proposed in [Bat05b]. The authors checked the feature models against a set of user-provided feature dependences of the form $F \rightarrow A \vee B$ for features F, A, and B. This approach was adopted by Czarnecki and Pietroszek [CP06] to verify software product lines modelled as feature-based model templates. The product line is represented as an UML specification whose elements are tagged with boolean expressions representing their presence in an instantiation. These boolean expressions correspond to the inclusion of a feature in a product specification. These templates typically have a set of well-formedness constraints which each instantiation should satisfy. In the spirit of [Bat05b], these constraints are converted to a propositional formula; feature models are then checked against this formula to make sure that they do not allow ill-formed template instantiations. These two approaches relied on user-provided constraints when validating feature models. When features are mapped to modules, the feature model used to describe a product line is a *module interconnection language* [PDN82].

Thüm et. al [TSKA11] consider proof composition in the verification of a Java-based software product line. Each product is annotated with invariants from which the Krakatoa/Why tool generates proof obligations to be verified in Coq. To avoid maintaining these proofs for each product, the authors maintain proof pieces in each feature and compose the pieces for an individual product. Their notion of composition is strictly syntactic: proof scripts are copied together to build the final proofs and have to be rechecked for each product. Importantly, features only add new premises and conjunctions to the conclusions of the obligations generated by Krakatoa/Why, allowing syntactic composition to work well for this application. As features begin to apply more subtle changes to definitions and proofs, it is not clear how to effectively syntactically glue together Coq's proof scripts. Using the abstraction mechanisms provided by Coq to implement features enables a more semantic notion of composition.

7.4.1 Modular Product Line Analyses

The modules with variability from Chapter 6 fit into the broader framework of Kastner's variability-aware modules [KOE12a]. The key difference is that we begin with an open set of modules. Once the product line is known and the variability is fixed, these open modules are lifted to variability-aware modules. One difference with Kastner's system is that we allow duplicate definitions to be merged during composition by restricting the inclusion of the conflicting modules.

One approach to taming combinatorics in product line testing is to eliminate "irrelevant" features– those that do not effect the outcome of a test [KBK12]. Each test is meant to (automatically) establish a property π, for example that an account cannot have a negative balance in banking product line. For each test, a static analysis partitions the features of the product line into relevant and irrelevant features. There is an implicit proof that "whether the test passes or fails is independent of whether an irrelevant feature is present or not". Assuming that features F and H are irrelevant to a test of π, establishing $\pi(\mathsf{F}+\mathsf{G}+\mathsf{H}+\mathsf{I})$ is equivalent to testing $\pi(\mathsf{F}+\mathsf{H}) \wedge \pi(\mathsf{G}+\mathsf{I})$. By the implicit proof $\pi(\mathsf{F}+\mathsf{H})$ is true, so this is equivalent to testing $\pi(\mathsf{G}+\mathsf{I})$. Maximizing modularity is key to taming combinatorics, but even this partial decomposition reduces the testing burden considerably.

Another approach uses the Alloy tool-set to generate product line tests [UKB10]. In this setting, the semantic property π for a product is an Alloy formula that is analyzed with Alloy to generate product tests. The specification of a product is the conjunction of the specifications of its constituent features, i.e. $+_\pi$ is $\wedge$. This observation allows $\pi(\mathsf{F}+\mathsf{G})$ to be decomposed to $\pi(\mathsf{F}) \wedge \pi(\mathsf{G})$. Even though Alloy now analyzes two formulas, each one is simpler. Furthermore, results from analyzing $\pi(\mathsf{F})$ are used to bound the analysis of $\pi(\mathsf{G})$, making the second search much more efficient. This dependence means that features are not truly analyzed in isolation (though they could be!), but speedups of up to 66X were gained nonetheless.

Chapter 8

Reflections and Conclusion

8.1 On the Importance of Engineering

The most common question about our approach to modular mechanized meta-theory is "How can I implement feature X?" The answer often depends on X, but it reveals an important truth: the adaptability of a feature module hinges on the abstractions it provides. If a module lacks the abstractions needed for an extension, it must be reengineered. Architecting product lines (sets of similar programs) has long existed in the software engineering community [McI68, Par76], as has the challenge of achieving object-oriented code reuse in this context [SB98, VN96]. The essence of reusable designs – be they code or proofs – is engineering. There is no magic bullet, but rather a careful trade-off between flexibility and specialization. A spectrum of common changes must be captured in the abstractions of a feature and its interface. The contributions of this dissertation highlight the importance of finding and supporting the right kinds of abstractions for the target domain.

- Chapter 2 demonstrated how the module system of the target domain can facilitate convenient abstractions. By default, object-oriented languages implicitly abstract classes over all possible subclasses, which is insufficient to modularize every feature. The mixins of LFJ introduced implicit abstraction over all possible superclasses, an abstraction which has proven useful in engineering feature-oriented software product lines.

- Chapters 3, 4, and 5 showed that selecting the right abstractions (functors and monads) through analysis of the target domain (e.g. programming languages) enables a wide range of useful extensions.

- Chapter 6 demonstrated that factoring out commonalities enables reasoning to be reused across an entire family of related systems.

8.2 Conclusion

The design of a complex system is naturally expressed as a combination of distinguishing features. Features are particularly useful when comparing two similar systems: distinct features can differentiate between variations of a configurable system and also identify the novelties of extensions. The connection between conceptual features and their implementation is often lost when building a system, making it difficult to change the conceptual design or to independently develop and combine extensions. The goal of Feature-Oriented Software Design is to maintain an explicit link between features and their implementation, allowing the implementation of a system to be synthesized directly from a specification of its features.

Feature-Oriented approaches to synthesis can be broadly classified as either projectional or compositional. Projectional approaches explicitly annotate the implementation of each feature à la #ifdefs in C. This approach is easy to layer on top of existing languages and to apply to legacy systems. Existing implementations must be directly modified to support new features, however. In contrast, compositional approaches maintain the relationship between features and their implementations by modularizing each feature into distinct components. Because features can cut across the modularity boundaries of the implementation domain, these components are also typically implemented in an extended language.

Importantly, both approaches are fundamentally syntactic — tools are used to map the source of programs in the extended language to programs in the implementation language. Projectional tools are preprocessors which discard anything not annotated with a selected feature, while compositional tools define operators for combining implementations. In both cases, the meaning of a feature's implementation is defined by the behavior of a tool. Lacking a proper semantics, the implementation of a feature cannot be understood in isolation. Type errors in Jak feature modules, as an example, are not detected until after they are composed with an existing Java program. In the case of mechanized metatheory proofs, extensions to definitions are applied and then existing proofs must be manually examined and repaired.

This dissertation showed that implementing features as modules with a proper semantics allows features to be reasoned about in isolation. This effort can be reused to reason about products built from that feature, allowing a product to be understood in terms of the features it includes. For some implementation domains, this requires new languages with modularity constructs that support novel kinds of abstractions. The LFJ calculus presented in Chapter 2, for example, included feature modules with class refinements (mixins) inspired by Jak. In contrast to the purely syntactic Jak feature modules, the LFJ type system checked feature modules in isolation. The proof of soundness for LFJ demonstrated that compositions of well-typed modules produce well-typed programs.

Even when the implementation domain has rich modularity constructs, building feature modules depends on using the right abstractions. Most theorem provers have powerful abstraction mechanisms, but the cut-paste-patch approach is the most common means of proof reuse for mechanized metatheory of language extensions. Reuse is particularly important in this domain, as proofs can take several man-years to develop [Ler09]. Chapter 3 demonstrated that with the right abstractions, the development of even a relatively complex extended language like FGJ can be effectively modularized in Coq, allowing fully mechanized language definitions and proofs to be synthesized from these modular components.

One common form of language extension is the addition of new syntactic values, typing rules, and reduction rules, which requires abstracting definitions over recursive occurrences. A least fixpoint operator is needed to close the inductive loop of these definitions, but a general definition cannot be built using Coq's inductive datatype mechanism. Chapter 4 showed how such an operator can be defined for datatypes implemented using a novel form of Church-encodings. These Church-encoded datatypes allow the syntactic and semantic domains of languages to be extended with new values. The chapter also presented a novel solution to a long-standing problem with reasoning over Church-encodings, enabling modular proof development over extensible datatypes.

Adding new pieces of syntax to a language can add not just new values to its semantic domain, but also new computational effects. Monads [Mog91, Wad92a] have long been used to structure the effects of semantic domains, and recent work [SO10] has shown how modular semantic definitions can be extended with new effects by abstracting them over a monad. Chapter 4 demonstrated how to reason with modular monadic semantics by using an extensible monadic typing predicate and decomposing soundness proofs into three separate lemmas.

Finally, relying on syntactic composition mechanisms makes it difficult to efficiently reason about a family of products synthesized from a set of features. Implementing features as mixin modules with proper semantics reduces reasoning about an entire product-line to a module configuration problem by allowing feature modules to be reasoned about through their import and export interfaces. To ensure that a property holds for the entire product line, Chapter 6 established that it suffices to show that the import interface of every valid configuration is satisfied. These interface checks can be reduced to checking the validity of a propositional formula, which SAT solvers can efficiently solve in practice.

Appendix A

Lightweight Java

f	field name
m	method name
var	term variable
dcl	name of derived class
oid	object identifier
j, k, l	index

terminals	$::=$	
	$\vert$	$($
	$\vert$	$)$
	$\vert$	$[$
	$\vert$	$]$
	$\vert$	$,$
	$\vert$	$/$
	$\vert$	$:$
	$\vert$	$\prec_1$
	$\vert$	$\prec$
	$\vert$	$\rightarrow$
	$\vert$	$\mapsto$
	$\vert$	$\longrightarrow^1$
	$\vert$	$\Longrightarrow$
	$\vert$	$=$

	$\mid$	$==$	
	$\mid$	$\neq$	
	$\mid$	$\vdash$	
	$\mid$	$\forall$	
	$\mid$	$\exists$	
	$\mid$	$\in$	
	$\mid$	$\notin$	
	$\mid$	$\vee$	
	$\mid$	$\wedge$	
	$\mid$	$\emptyset$	
	$\mid$	$\cap$	
	$\mid$	$\perp$	'is disjoint from'
$x,\ y$	::=		term variable
	$\mid$	*var*	normal variable
	$\mid$	**this**	keyword
cl	::=		class name
	$\mid$	*dcl*	name of derived class
	$\mid$	**Object**	name of base of all classes
cl_{opt}	::=		class name option
	$\mid$	$H\,(oid)$ M	dynamic type lookup
fd	::=		field declaration
	$\mid$	$\tau\, f$	type and field name

fds	$::=$			field declarations
	$\mid$	$fd_1 \,..\, fd_k$		
$\overline{f}$	$::=$			list of fields
	$\mid$	$f_1 \,..\, f_k$	M	
	$\mid$	$\mathbf{fields}\,(P,\ cl)$	M	recursive fields lookup
s	$::=$			statement
	$\mid$	$var = \mathbf{new}\ cl\,();$		object construction
	$\mid$	$var = x\,;$		variable assignment
	$\mid$	$var = x\,.\,f\,;$		field read
	$\mid$	$x\,.\,f = y\,;$		field write
	$\mid$	$var = x\,.\,m\,(y_1 \,..\, y_k)\,;$		dynamically dispatched method call
	$\mid$	$\mathbf{if}\,(x == y)\ s\ \mathbf{else}\ s'$		conditional branch
	$\mid$	$\{\ s_1 \,..\, s_k\ \}$		
$\overline{s}$	$::=$			list of statements
	$\mid$	$[]$		empty
	$\mid$	s		singleton
	$\mid$	$\overline{s}_1 \,..\, \overline{s}_k$		concat
	$\mid$	$\theta\,(\overline{s})$	M	variable replacement
vd	$::=$			variable declaration
	$\mid$	$\tau\ var$		type and variable name
ms	$::=$			method signature

	$\vert$	$\tau\ m\ (vd_1,\ ..,\ vd_k)$		
mb	$::=$			method body
	$\vert$	$\overline{s}\ \mathbf{return}\ x\ ;$		
md	$::=$			method definition
	$\vert$	$ms\ \{\ mb\ \}$		
$menv$	$::=$			method environment
	$\vert$	$var_1\ ;\ ..\ var_k\ ;\ mb$		parameter names and body
	$\vert$	$\mathbf{find\_menv}\ (P,\ cl,\ m)$	M	lookup
$\overline{m}$	$::=$			list of method names
	$\vert$	$\mathbf{methods_1}(P,\ cl)$	M	methods lookup
	$\vert$	$\mathbf{methods}\ (P,\ cl)$	M	recursive methods lookup
cld	$::=$			class definition
	$\vert$	$\mathbf{class}\ dcl\ \mathbf{extends}\ cl\ \{\ fds\ md_1\ ..\ md_k\ \}$		
P	$::=$			program
	$\vert$	$cld_1\ ..\ cld_k$		
τ	$::=$			type
	$\vert$	cl		
τ_{opt}	$::=$			result of type lookup
	$\vert$	τ		lifted type

	$\mid$	$\mathbf{ftype}\,(P,\ cl,\ f)$	M	lookup type of a field recursively
	$\mid$	$\Gamma\,(x)$	M	type environment lookup
	$\mid$	cl_{opt}	M	class name option to type option conversion
$\overline{\tau}$	$::=$			types
	$\mid$	$\tau_1\,..\,\tau_k$		
π	$::=$			method type
	$\mid$	$\overline{\tau}\ \rightarrow\ \tau$		definition
	$\mid$	$\mathbf{mtype}\,(P,\ cl,\ m)$	M	type lookup
Γ	$::=$			type environment ($x \rightharpoonup \tau$)
	$\mid$	$[x\ \mapsto\ \tau]$		maps variable to type
	$\mid$	$\Gamma_1\,..\,\Gamma_k$	M	composes many
θ	$::=$			variable mapping ($x \rightharpoonup y$)
	$\mid$	$[y\,/\,x]$	M	y replaces x
	$\mid$	$\theta_1\,..\,\theta_k$	M	composes many
$v,\ w$	$::=$			value
	$\mid$	**null**		null value
	$\mid$	*oid*		object identifier
v_{opt}	$::=$			result of value lookup
	$\mid$	v		lifted value
	$\mid$	$L\,(x)$	M	dynamic value lookup

	$\mid$	$H\,(oid,\,f)$	M	dynamic value lookup
L	::=			variable state $(x \rightharpoonup v)$
	$\mid$	$[x\ \mapsto\ v]$	M	x maps to v
	$\mid$	$L_1\,..\,L_k$	M	composes many
H	::=			heap $(oid \rightharpoonup (cl * (f \rightharpoonup v)))$
	$\mid$	$H\,[oid\ \mapsto\ (cl,\,f_1\ \mapsto\ v_1\,..\,f_k\ \mapsto\ v_k)]$	M	new oid of type cl in H
	$\mid$	$H\,[(oid,\,f)\ \mapsto\ v]$	M	f of oid mapping to v in H
$config$	::=			configuration
	$\mid$	$(L,\,H,\,\overline{s})$		$\overline{s}$ to execute under L and H
	$\mid$	$(L,\,H,\,\mathbf{NPE})$		null pointer exception
$formula$	::=			formulas
	$\mid$	$judgement$		judgement
	$\mid$	$formula_1\quad ..\quad formula_k$		
	$\mid$	$(formula)$		bracketed
	$\mid$	$formula \vee formula'$		or
	$\mid$	$formula \wedge formula'$		and
	$\mid$	$P = P'$		program alias
	$\mid$	$\overline{f} = \overline{f}'$		fields alias
	$\mid$	$\overline{m} = \overline{m}'$		method name aliases
	$\mid$	$menv = menv'$		method environment alias
	$\mid$	$\tau_{opt} = {\tau_{opt}}'$		type option alias
	$\mid$	$\pi = \pi'$		method type alias

$\mid$	$\theta = \theta'$	mapping alias
$\mid$	$L = L'$	stack alias
$\mid$	$H = H'$	heap alias
$\mid$	$\Gamma = \Gamma'$	type environment alias
$\mid$	$fds = fds'$	field declarations alias
$\mid$	$v_{opt} = v$	value lookup alias
$\mid$	$v == w$	value equality
$\mid$	$v \neq w$	value inequality
$\mid$	$\theta\,(x) = y$	variable lookup
$\mid$	$L\,(x) = L'\,(y)$	equal values
$\mid$	$L\,(x) \neq L'\,(y)$	not equal values
$\mid$	$x_1 \mathinner{..} x_k \perp y_1 \mathinner{..} y_j$	disjoint variable lists
$\mid$	$x_1 \mathinner{..} x_k \perp \mathbf{dom}\,(L)$	disjoint variables and domain
$\mid$	$\overline{f} \perp \overline{f}'$	disjoint field sets
$\mid$	$cl \in \mathbf{names}\,(P)$	class name defined in program
$\mid$	$\mathbf{distinct}\,(f_1 \mathinner{..} f_k)$	distinct field names
$\mid$	$\mathbf{distinct}\,(\mathbf{names}\,(P))$	distinct names of class definitions
$\mid$	$cld \in P$	class definition in program
$\mid$	$m \notin \overline{m}$	m not in $\overline{m}$
$\mid$	$oid \notin \mathbf{dom}\,(H)$	oid not in domain of H
$\mid$	$\forall x \in \mathbf{dom}\,(L)\,.\,(formula)$	for all variables in domain of L
$\mid$	$\forall f \in \overline{f}\,.\,(formula)$	for all fields in $\overline{f}$
$\mid$	$\forall m \in \overline{m}\,.\,(formula)$	for all method names in $\overline{m}$
$\mid$	$\forall oid \in \mathbf{dom}\,(H)\,.\,(formula)$	for all objects in domain of H

	$\mid$	$\exists cl\,.\,(\mathit{formula})$	there exists a cl such that *formula*
	$\mid$	$\exists \tau\,.\,(\mathit{formula})$	there exists a τ such that *formula*
subtyping	$::=$		
	$\mid$	$P \vdash \tau \prec_1 \tau'$	direct normal subtyping
	$\mid$	$P \vdash \tau \prec \tau'$	normal subtyping
	$\mid$	$P \vdash \overline{\tau} \prec \overline{\tau}'$	normal, multiple subtyping
	$\mid$	$P \vdash \tau_{opt} \prec {\tau_{opt}}'$	option subtyping
well_formedness	$::=$		
	$\mid$	$P \vdash \tau$	well-formed type
	$\mid$	$P, \Gamma \vdash s$	well-formed statement
	$\mid$	$\vdash_\tau md$	well-formed method
	$\mid$	$P \vdash cld$	well-formed class
	$\mid$	$\vdash P$	well-formed program
	$\mid$	$P, H \vdash v_{opt} \prec \tau_{opt}$	well-formed value
	$\mid$	$P, \Gamma, H \vdash L$	well-formed stack
	$\mid$	$P \vdash H$	well-formed heap
	$\mid$	$P, \Gamma \vdash \mathit{config}$	well-formed configuration
smallstep	$::=$		
	$\mid$	$\mathit{config} \longrightarrow^1_P \mathit{config}'$	one step reduction in P
judgement	$::=$		
	$\mid$	*subtyping*	
	$\mid$	*well_formedness*	

	$\vert$	$smallstep$
$user\_syntax$	$::=$	
	$\vert$	f
	$\vert$	m
	$\vert$	var
	$\vert$	dcl
	$\vert$	oid
	$\vert$	j
	$\vert$	$terminals$
	$\vert$	x
	$\vert$	cl
	$\vert$	cl_{opt}
	$\vert$	fd
	$\vert$	fds
	$\vert$	$\overline{f}$
	$\vert$	s
	$\vert$	$\overline{s}$
	$\vert$	vd
	$\vert$	ms
	$\vert$	mb
	$\vert$	md
	$\vert$	$menv$
	$\vert$	$\overline{m}$
	$\vert$	cld

$|\quad P$

$|\quad \tau$

$|\quad \tau_{opt}$

$|\quad \overline{\tau}$

$|\quad \pi$

$|\quad \Gamma$

$|\quad \theta$

$|\quad v$

$|\quad v_{opt}$

$|\quad L$

$|\quad H$

$|\quad \mathit{config}$

$|\quad \mathit{formula}$

$\boxed{P \vdash \tau \prec_1 \tau'}$ direct normal subtyping

$$\frac{\textbf{class}\ \mathit{dcl}\ \textbf{extends}\ \mathit{cl}\ \{\ \mathit{fds}\ \overline{\mathit{md}_k}^{\,k}\ \} \in P}{P \vdash \mathit{dcl} \prec_1 \mathit{cl}} \quad \text{STY\_DIR}$$

$\boxed{P \vdash \tau \prec \tau'}$ normal subtyping

$$\frac{P \vdash \tau \prec_1 \tau'}{P \vdash \tau \prec \tau'} \quad \text{STY\_FROM\_DIRECT}$$

$$\frac{}{P \vdash \tau \prec \tau} \quad \text{STY\_REFLEXIVE}$$

$$\frac{\begin{array}{c} P \vdash \tau \prec \tau' \\ P \vdash \tau' \prec \tau'' \end{array}}{P \vdash \tau \prec \tau''} \quad \text{STY\_TRANSITIVE}$$

$\boxed{P \vdash \overline{\tau} \prec \overline{\tau}'}$ normal, multiple subtyping

$$\frac{\overline{P \vdash \tau_k \prec \tau'_k}^{\,k}}{P \vdash \overline{\tau_k}^{\,k} \prec \overline{\tau'_k}^{\,k}} \quad \text{STY\_MANY}$$

$\boxed{P \vdash \tau_{opt} \prec \tau_{opt}{}'}$ option subtyping

$$\frac{\begin{array}{l}\tau_{opt} = \tau \\ \tau_{opt}{}' = \tau' \\ P \vdash \tau \prec \tau'\end{array}}{P \vdash \tau_{opt} \prec \tau_{opt}{}'} \quad \text{STY\_OPTION}$$

$\boxed{P \vdash \tau}$ well-formed type

$$\frac{cl \in \mathbf{names}\,(P)}{P \vdash cl} \quad \text{WF\_VALID\_DCL}$$

$\boxed{P, \Gamma \vdash s}$ well-formed statement

$$\frac{P \vdash cl \prec \Gamma\,(var)}{P, \Gamma \vdash var = \mathbf{new}\; cl\,();} \quad \text{WF\_NEW}$$

$$\frac{P \vdash \Gamma\,(x) \prec \Gamma\,(var)}{P, \Gamma \vdash var = x\,;} \quad \text{WF\_VAR\_ASSIGN}$$

$$\frac{\begin{array}{l}\Gamma\,(x) = cl \\ \mathbf{ftype}\,(P,\ cl,\ f) = \tau \\ P \vdash \tau \prec \Gamma\,(var)\end{array}}{P, \Gamma \vdash var = x\,.\,f\,;} \quad \text{WF\_FIELD\_READ}$$

$$\frac{\begin{array}{l}\Gamma\,(x) = cl \\ \mathbf{ftype}\,(P,\ cl,\ f) = \tau \\ P \vdash \Gamma\,(y) \prec \tau\end{array}}{P, \Gamma \vdash x\,.\,f = y\,;} \quad \text{WF\_FIELD\_WRITE}$$

$$\frac{\begin{array}{l}\Gamma\,(x) = cl \\ \mathbf{mtype}\,(P,\ cl,\ m) = \overline{\tau_k}^{\,k} \ \rightarrow\ \tau' \\ \overline{P\ \vdash\ \Gamma\,(y_k) \prec \tau_k}^{\,k} \\ P\ \vdash\ \tau' \prec \Gamma\,(var)\end{array}}{P,\,\Gamma\ \vdash\ var = x\,.\,m\,(\overline{y_k}^{\,k})\,;}\quad \text{WF\_MCALL}$$

$$\frac{\begin{array}{l}P\ \vdash\ \Gamma\,(x) \prec \Gamma\,(y) \vee P\ \vdash\ \Gamma\,(x) \prec \Gamma\,(y) \\ P,\,\Gamma\ \vdash\ s_1 \\ P,\,\Gamma\ \vdash\ s_2\end{array}}{P,\,\Gamma\ \vdash\ \mathbf{if}\,(x == y)\ s_1\ \mathbf{else}\ s_2}\quad \text{WF\_IF}$$

$$\frac{\overline{P,\,\Gamma\ \vdash\ s_k}^{\,k}}{P,\,\Gamma\ \vdash\ \{\,\overline{s_k}^{\,k}\,\}}\quad \text{WF\_BLOCK}$$

$\boxed{\vdash_\tau\ md}$ well-formed method

$$\frac{\begin{array}{l}\overline{P\ \vdash\ \tau_k}^{\,k} \\ \Gamma = [\mathbf{this}\ \mapsto\ cl]\,\overline{[var_k\ \mapsto\ \tau_k]}^{\,k} \\ \overline{P,\,\Gamma\ \vdash\ s_l}^{\,l} \\ P\ \vdash\ \Gamma\,(x) \prec \tau\end{array}}{\vdash_{cl}\ \tau\ m\,(\overline{\tau_k\ var_k}^{\,k})\,\{\,\overline{s_l}^{\,l}\ \mathbf{return}\ x\,;\ \}}\quad \text{WF\_METHOD}$$

$\boxed{P\ \vdash\ cld}$ well-formed class

$$\frac{\begin{array}{l} P \vdash cl \\ fds = \overline{\tau_j\, f_j}^{\,j} \\ \mathbf{distinct}\,(\overline{f_j}^{\,j}) \\ \overline{f_j}^{\,j} \perp \mathbf{fields}\,(P,\ cl) \\ \overline{P \vdash \tau_j}^{\,j} \\ \overline{\vdash_{dcl}\ md_k}^{\,k} \\ \mathbf{methods_1}(P,\ dcl) = \overline{m} \\ \mathbf{methods}\,(P,\ cl) = \overline{m}' \\ \forall m \in \overline{m}' \,.\, (m \notin \overline{m} \vee \mathbf{mtype}\,(P,\ dcl,\ m) = \mathbf{mtype}\,(P,\ cl,\ m)) \end{array}}{P \vdash \mathbf{class}\ dcl\ \mathbf{extends}\ cl\ \{\, fds\ \overline{md_k}^{\,k}\ \}} \quad \text{WF\_CLASS}$$

$\boxed{\vdash P}$ well-formed program

$$\frac{\begin{array}{l} P = \overline{cld_k}^{\,k} \\ \mathbf{distinct}\,(\mathbf{names}\,(P)) \\ \overline{P \vdash cld_k}^{\,k} \end{array}}{\vdash P} \quad \text{WF\_PROGRAM}$$

$\boxed{P,\ H \vdash v_{opt} \prec \tau_{opt}}$ well-formed value

$$\frac{\tau_{opt} = \tau}{P,\ H \vdash \mathbf{null} \prec \tau_{opt}} \quad \text{WF\_NULL}$$

$$\frac{P \vdash H\,(oid) \prec \tau_{opt}}{P,\ H \vdash oid \prec \tau_{opt}} \quad \text{WF\_OBJECT}$$

$\boxed{P,\ \Gamma,\ H \vdash L}$ well-formed stack

$$\frac{\forall x \in \mathbf{dom}\,(L) \,.\, (\exists \tau \,.\, (\Gamma\,(x) = \tau \wedge P \vdash \tau \wedge P,\ H \vdash L\,(x) \prec \tau))}{P,\ \Gamma,\ H \vdash L} \quad \text{WF\_STACK}$$

$\boxed{P \vdash H}$ well-formed heap

$$\frac{\begin{array}{l}\forall oid \in \mathbf{dom}\,(H)\,.\\ \quad (\exists cl\,.\,(H\,(oid) = cl \wedge P \vdash cl \wedge\\ \quad \forall f \in \mathbf{fields}\,(P,\ cl)\,.\\ \qquad (\exists \tau\,.\,(\mathbf{ftype}\,(P,\ cl,\ f) = \tau \wedge P,\ H \vdash H\,(oid,\ f) \prec \tau))))\end{array}}{P \vdash H} \quad \text{WF\_HEAP}$$

$\boxed{P,\ \Gamma \vdash \mathit{config}}$ well-formed configuration

$$\frac{\begin{array}{l}P \vdash H\\ P,\ \Gamma,\ H \vdash L\\ \overline{P,\ \Gamma \vdash s_k}^{\,k}\end{array}}{P,\ \Gamma \vdash (L,\ H,\ \overline{s_k}^{\,k})} \quad \text{WF\_ALL}$$

$$\frac{}{P,\ \Gamma \vdash (L,\ H,\ \mathbf{NPE})} \quad \text{WF\_ALL\_NPE}$$

$\boxed{\mathit{config} \longrightarrow^1_P \mathit{config}'}$ one step reduction in P

$$\frac{L\,(x) = \mathbf{null}}{(L,\ H,\ var = x\,.\,f\,;\,\overline{s}) \longrightarrow^1_P (L,\ H,\ \mathbf{NPE})} \quad \text{FIELD\_READ\_NPE}$$

$$\frac{L\,(x) = \mathbf{null}}{(L,\ H,\ x\,.\,f = y\,;\,\overline{s}) \longrightarrow^1_P (L,\ H,\ \mathbf{NPE})} \quad \text{FIELD\_WRITE\_NPE}$$

$$\frac{L\,(x) = \mathbf{null}}{(L,\ H,\ var = x\,.\,m\,(\overline{y_k}^{\,k})\,;\,\overline{s}) \longrightarrow^1_P (L,\ H,\ \mathbf{NPE})} \quad \text{MCALL\_NPE}$$

$$\frac{\begin{array}{l}oid \notin \mathbf{dom}\,(H)\\ \mathbf{fields}\,(P,\ cl) = \overline{f_k}^{\,k}\\ H' = H\,[oid \mapsto (cl,\ \overline{f_k \mapsto \mathbf{null}}^{\,k})]\end{array}}{(L,\ H,\ var = \mathbf{new}\ cl\,();\overline{s}) \longrightarrow^1_P (L\,[var \mapsto oid],\ H',\ \overline{s})} \quad \text{NEW}$$

$$\frac{L\,(x) = v}{(L,\ H,\ \mathit{var} = x\,;\ \overline{s}) \longrightarrow_P^1 (L\,[\mathit{var}\ \mapsto\ v],\ H,\ \overline{s})} \quad \text{VAR\_ASSIGN}$$

$$\frac{\begin{array}{c} L\,(x) = \mathit{oid} \\ H\,(\mathit{oid},\ f) = v \end{array}}{(L,\ H,\ \mathit{var} = x\,.\,f\,;\ \overline{s}) \longrightarrow_P^1 (L\,[\mathit{var}\ \mapsto\ v],\ H,\ \overline{s})} \quad \text{FIELD\_READ}$$

$$\frac{\begin{array}{c} L\,(x) = \mathit{oid} \\ L\,(y) = v \end{array}}{(L,\ H,\ x\,.\,f = y\,;\ \overline{s}) \longrightarrow_P^1 (L,\ H\,[(\mathit{oid},\ f)\ \mapsto\ v],\ \overline{s})} \quad \text{FIELD\_WRITE}$$

$$\frac{\begin{array}{l} \mathbf{find\_menv}\,(P,\ cl,\ m) = \overline{\mathit{var}_k;}^{\,k}\ \overline{s}'\ \mathbf{return}\,x'\,; \\ \overline{\mathit{var}'_k}^{\,k} \perp \mathbf{dom}\,(L) \\ \overline{L\,(y_k) = v_k}^{\,k} \\ L' = L\,\overline{[\mathit{var}'_k\ \mapsto\ v_k]}^{\,k} \\ \theta = [x\,/\,\mathbf{this}]\ \overline{[\mathit{var}'_k\,/\,\mathit{var}_k]}^{\,k} \\ \theta\,(x') = y \end{array}}{(L,\ H,\ \mathit{var} = x\,.\,m\,(\overline{y_k}^{\,k})\,;\ \overline{s}) \longrightarrow_P^1 (L',\ H,\ \theta\,(\overline{s}')\ \mathit{var} = y\,;\ \overline{s})} \quad \text{MCALL}$$

$$\frac{\begin{array}{c} L\,(x) = v \\ L\,(y) = w \\ v == w \end{array}}{(L,\ H,\ \mathbf{if}\,(x == y)\ s_1\ \mathbf{else}\ s_2\ \overline{s}) \longrightarrow_P^1 (L,\ H,\ s_1\,\overline{s})} \quad \text{IF\_TRUE}$$

$$\frac{\begin{array}{c} L\,(x) = v \\ L\,(y) = w \\ v\ \neq\ w \end{array}}{(L,\ H,\ \mathbf{if}\,(x == y)\ s_1\ \mathbf{else}\ s_2\ \overline{s}) \longrightarrow_P^1 (L,\ H,\ s_2\,\overline{s})} \quad \text{IF\_FALSE}$$

$$\frac{}{(L,\ H,\ \{\ s_1 \mathinner{..} s_k\ \}\ \overline{s}) \longrightarrow_P^1 (L,\ H,\ \ (s_1 \mathinner{..} s_k)\ \overline{s})} \quad \text{BLOCK}$$

Bibliography

[A$^+$05] B.E. Aydemir et al. Mechanized Metatheory for the Masses: The PoplMark Challenge. In *TPHOLs'05*, 2005.

[AD05] Davide Ancona and Sophia Drossopoulou. Polymorphic bytecode: Compositional compilation for java-like languages. In *In ACM Symp. on Principles of Programming Languages 2005*. ACM Press, 2005.

[AH07] Sven Apel and DeLesley Hutchins. An overview of the gDEEP calculus. Technical Report MIP-0712, Department of Informatics and Mathematics, University of Passau, November 2007.

[AKL08] Sven Apel, Christian Kästner, and Christian Lengauer. Feature Featherweight Java: A calculus for feature-oriented programming and stepwise refinement. In *GPCE '08: Proceedings of the 7th International Conference on Generative Programming and Component Engineering*. ACM Press, October 2008.

[AKL09] Sven Apel, Christian Kästner, and Christian Lengauer. Featurehouse: Language-independent, automated software composition. In *ICSE*, 2009.

[ARW$^+$13] Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Grösslinger, and Dirk Beyer. Strategies for product-line verification: case studies and experiments. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 482–491, 2013.

[Bas87] Paul Bassett. Frame-based software engineering. *IEEE Software*, 4(4), 1987.

[Bat04] D. Batory. Feature-oriented programming and the AHEAD tool suite. *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 702–703, May 2004.

[Bat05a] Don Batory. Feature models, grammars, and propositional formulas. *Software Product Lines*, pages 7–20, 2005.

[Bat05b] Don Batory. Feature models, grammars, and propositional formulas. In *In Software Product Lines Conference, LNCS 3714*, pages 7–20. Springer, 2005.

[BB85] C. Böhm and A. Berarducci. Automatic synthesis of typed lambda-programs on term algebras. *Theor. Comput. Sci.*, 39, 1985.

[BB08] Don Batory and Egon Boerger. Modularizing theorems for software product lines: The jbook case study. *Jour. of Universal Computer Science*, July 2008.

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In *OOPSLA*, pages 303–311, 1990.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development.* Springer-Verlag, Berlin, 2004.

[BCS00] D. Batory, Rich Cardone, and Y. Smaragdakis. Object-oriented frameworks and product-lines. In *SPLC*, 2000.

[BH11] Patrick Bahr and Tom Hvitved. Compositional data types. In *Proceedings of the seventh ACM SIGPLAN workshop on Generic programming*, WGP '11, pages 83–94. ACM, 2011.

[BHK11] Don Batory, Peter Höfner, and Jongwook Kim. Feature interactions, products, and composition. In *GPCE*, pages 13–22. ACM, 2011.

[BHMZ13] Don Batory, Peter Hofner, Bernhard Moeller, and Andreas Zeland. Features, modularity, and variation points. In *Workshop on Feature-Oriented Software Development (FOSD) 2013*, FOSD '13. ACM, 2013.

[Big94] Ted J. Biggerstaff. The library scaling problem and the limits of concrete component reuse. In *Third International Conference on Reuse*, 1994.

[BKH11] D. Batory, J. Kim, and P. Höfner. Feature interactions, products, and composition. In *GPCE*, 2011.

[BL92] Gilad Bracha and Gary Lindstrom. Modularity meets inheritance. In *IN PROC. INTERNATIONAL CONFERENCE ON COMPUTER LANGUAGES*, pages 282–290. IEEE Computer Society, 1992.

[Boi04] O. Boite. Proof reuse with extended inductive types. In *Theorem Proving in Higher Order Logics*, pages 50–65, 2004.

[BP12] Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *CoRR*, abs/1203.1539, 2012.

[BSST93] Don Batory, Vivek Singhal, Marty Sirkin, and Jeff Thomas. Scalable software libraries. In *Proceedings of the 1st ACM SIGSOFT symposium on Foundations of software engineering*, SIGSOFT '93, pages 191–199, New York, NY, USA, 1993. ACM.

[C28] Ibm system/360 operating system introduction.

[CDKD86] D. Clément, T. Despeyroux, G. Kahn, and J. Despeyroux. A Simple Applicative Language: mini-ML. In *LFP '86*, 1986.

[CH86] T. Coquand and Gérard Huet. The calculus of constructions. Technical Report RR-0530, INRIA, May 1986.

[Chl08] A. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *ICFP'08*, 2008.

[Chl10] Adam Chlipala. A verified compiler for an impure functional language. In *POPL 2010*, January 2010.

[CM93] Pietro Cenciarelli and Eugenio Moggi. A syntactic approach to modularity in denotational semantics. In *In Proceedings of the Conference on Category Theory and Computer Science*, CCTCS '93, 1993.

[Con79] Control Data Corporation. *ALGOL-60 version 5 reference manual*, 5 edition, 1979.

[Coo89] William Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, 1989.

[Coo91] William R. Cook. Object-oriented programming versus abstract data types. In *Foundations of Object-Oriented Languages*. Springer-Verlag, 1991.

[CP06] Krzysztof Czarnecki and Krzysztof Pietroszek. Verifying feature-based model templates against well-formedness ocl constraints. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*. ACM Press, 2006.

[Cur34] H. B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences of the United States of America*, 20(11):pp. 584–590, 1934.

[Del00] David Delahaye. A tactic language for the system coq. In *Proceedings of Logic for Programming and Automated Reasoning (LPAR), Reunion Island, volume 1955 of LNCS*, pages 85–95. Springer, 2000.

[Fil99] Andrzej Filinski. Representing layered monads. In *Proceedings of the 26th*

ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '99, pages 175–188. ACM, 1999.

[Fil07] Andrzej Filinski. On the relations between monadic semantics. *Theor. Comput. Sci.*, 375(1-3):41–75, 2007.

[Fil10] Andrzej Filinski. Monads in action. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '10, pages 483–494. ACM, 2010.

[FKA$^+$13] Janet Feigenspan, Christian Kästner, Sven Apel, Jörg Liebig, Michael Schulze, Raimund Dachselt, Maria Papendieck, Thomas Leich, and Gunter Saake. Do background colors improve program comprehension in the #ifdef hell? *Empirical Softw. Engg.*, 18(4):699–745, August 2013.

[GH11] J. Gibbons and R. Hinze. Just do it: simple monadic equational reasoning. In *ICFP '11*, 2011.

[GZND11] G. Gonthier, B. Ziliani, A. Nanevski, and D. Dreyer. How to make ad hoc proof automation less ad hoc. In *ICFP '11*, 2011.

[HF08] Graham Hutton and Diana Fulger. Reasoning about effects: Seeing the wood through the trees. In *Proceedings of the Ninth Symposium on Trends in Functional Programming*, 2008.

[Hin05] R. Hinze. Church numerals, twice! *JFP*, 15(1):1–13, 2005.

[How80] William A. Howard. The formulas-as-types notion of construction. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, 1980.

[Hue97] Gérard Huet. The zipper. *J. Funct. Program.*, 7(5):549–554, 1997.

[Huf12] Brian Huffman. Formal verification of monad transformers. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, ICFP '12, pages 15–16. ACM, 2012.

[Hug00] John Hughes. Generalising monads to arrows. *Sci. Comput. Program.*, 37(1-3):67–111, 2000.

[Hut99] G. Hutton. A tutorial on the universality and expressiveness of fold. *J. Funct. Program.*, 9(4):355–372, 1999.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

[Jan10] Mikoláš Janota. *SAT Solving in Interactive Configuration.* PhD thesis, University College Dublin, November 2010.

[Jas11] Mauro Jaskelioff. Monatron: An extensible monad transformer library. In *Implementation and Application of Functional Languages*, volume 5836 of *Lecture Notes in Computer Science*, pages 233–248. Springer, 2011.

[JD93] Mark P. Jones and Luc Duponcheel. Composing monads. Research Report YALEU/DCS/RR-1004, Yale University, 1993.

[KAK08] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 311–320, 2008.

[Kan05] K.C. Kang. Private Correspondence, 2005.

[KBK12] Chang Hwan Peter Kim, Don S. Batory, and Sarfraz Khurshid. Reducing combinatorics in testing product lines. In *AOSD*, 2012.

[KGR$^+$11] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 805–824, New York, NY, 2011.

[KLO12] Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In *The 1st ACM SIGPLAN Workshop on Higher-Order Programming with Effects*, HOPE '12, 2012.

[KM] Matt Kaufmann and J Strother Moore. An ACL2 tutorial.

[KOE12a] Christian Kästner, Klaus Ostermann, and Sebastian Erdweg. A variability-aware module system. In *OOPSLA*, 2012.

[KOE12b] Christian Kästner, Klaus Ostermann, and Sebastian Erdweg. A variability-aware module system. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 773–792, New York, NY, 2012.

[KS13] Steven Keuchel and Tom Schrijvers. Generic datatypes à la carte. In *Proceedings of the 9th ACM SIGPLAN workshop on Generic programming*, WGP '13. ACM, 2013.

[Lee89] Peter Lee. *Realistic Compiler Generation*. Foundations of Computing. MIT Press, 1989.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52:107–115, July 2009.

[Lev06] Paul Blain Levy. Monads and adjunctions for global exceptions. *Electron. Notes Theor. Comput. Sci.*, 158:261–287, 2006.

[LH96] S. Liang and P. Hudak. Modular denotational semantics for compiler construction. In *ESOP '96*, 1996.

[LHBL06] Roberto Lopez-Herrejon, Don Batory, and Christian Lengauer. A Disciplined Approach to Aspect Composition. In *PEPM*, 2006.

[LHJ95] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *POPL '95*, 1995.

[LP03] M. Y. Levin and B. C. Pierce. Tinkertype: A language for playing with formal systems. *Journal of Functional Programming*, 13(2), March 2003.

[Mac84] D. MacQueen. Modules for standard ML. In *LFP '84*, 1984.

[McI68] M. D. McIlroy. Mass-produced software components. *Proc. NATO Conf. on Software Engineering, Garmisch, Germany*, 1968.

[MH01] R. Monson-Haefel. *Enterprise Java Beans.* O'Reilly, 3rd edition, 2001.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Mog89] Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Edinburgh University, Department of Computer Science, June 1989.

[Mog91] E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1), July 1991.

[Mos92] Peter D. Mosses. *Action Semantics*, volume 26 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1992.

[Mos04] Peter D. Mosses. Modular structural operational semantics. *J. Log. Algebr. Program.*, 60-61:195–228, 2004.

[MP08] Conor Mcbride and Ross Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, 2008.

[Mul06] A. Mulhern. Proof weaving. In *WMM '06*, September 2006.

[Nor07] Ulf Norell. Towards a practical programming language based on dependent type theory, 2007.

[NWP02] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: a proof assistant for higher-order logic.* Springer-Verlag, Berlin, Heidelberg, 2002.

[OSC10] Bruno C. d. S. Oliveira, Tom Schrijvers, and William R. Cook. EffectiveAdvice: disciplined advice with explicit effects. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, AOSD '10, pages 109–120. ACM, 2010.

[Par76] D.L. Parnas. On the design and development of program families. *IEEE TSE*, SE-2(1):1 – 9, March 1976.

[PDN82] R. Prieto-Diaz and James Neighbors. Module interconnection languages: A survey. Technical report, University of California at Irvine, August 1982. ICS Technical Report 189.

[Pie02] B. C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[PM93] C. Paulin-Mohring. Inductive definitions in the system Coq - rules and properties. In *TLCA '93*, 1993.

[Pol98] Robert Pollack. How to believe a machine-checked proof. In *Twenty Five Years of Constructive Type Theory*, 1998.

[PP02] Gordon D. Plotkin and John Power. Notions of computation determine monads. In *Proceedings of the 5th International Conference on Foundations of Software*

Science and Computation Structures, FoSSaCS '02, pages 342–356. Springer-Verlag, 2002.

[PP09] Gordon Plotkin and Matija Pretnar. Handlers of algebraic effects. In *Programming Languages and Systems: 18th European Symposium on Programming, ESOP 2009*, volume 5502 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 2009.

[PPM90] F. Pfenning and C. Paulin-Mohring. Inductively defined types in the calculus of constructions. In *MFPS V*, 1990.

[Rey83] John C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.

[Rey94] John C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In *Theoretical aspects of object-oriented programming.* MIT Press, 1994.

[S$^+$07] Peter Sewell et al. Ott: effective tool support for the working semanticist. In *ICFP '07*, 2007.

[SB98] Yannis Smaragdakis and Don Batory. Implementing reusable object-oriented components. In *In the 5th Int. Conf. on Software Reuse (ICSR 98*, pages 36–45. Society Press, 1998.

[SNO$^+$07] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: effective tool support for the working semanticist. In *ICFP '07: Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, pages 1–12, New York, 2007. ACM.

[SO08] M. Sozeau and N. Oury. First-class type classes. In *TPHOLs '08*, 2008.

[SO10] Tom Schrijvers and Bruno C. d. S. Oliveira. The monad zipper. Report CW 595, Dept. of Computer Science, K.U.Leuven, 2010.

[SS12] Christopher Schwaab and Jeremy G. Siek. Modular type-safety proofs using dependent types. *CoRR*, abs/1208.0535, 2012.

[SSB01] Robert Stärk, Joachim Schmid, and Egon Börger. Java and the Java Virtual Machine - definition, verification, validation, 2001.

[SSP07] Rok Strnisa, Peter Sewell, and Matthew J. Parkinson. The Java module system: core design and semantic definition. In *OOPSLA*, pages 499–514, 2007.

[Ste94] Guy L. Steele, Jr. Building interpreters by composing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '94, pages 472–492. ACM, 1994.

[Sto77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[Swi08] W. Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4), 2008.

[TBKC07] Sahil Thaker, Don Batory, David Kitchin, and William Cook. Safe composition of product lines. In *GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering*, pages 95–104, New York, NY, USA, 2007. ACM.

[TSKA11] T. Thüm, I. Schaefer, M. Kuhlemann, and S. Apel. Proof composition for deductive verification of software product lines. In *Software Testing, Verification and Validation Workshops (ICSTW) 2011*, pages 270 –277, march 2011.

[UKB10] E. Uzuncaova, S. Khurshid, and D. Batory. Incremental test generation for software product lines. *Software Engineering, IEEE Transactions on*, may-june 2010.

[UV00] T. Uustalu and V. Vene. Coding recursion a la Mendler. In *WGP '00*, pages 69–85, 2000.

[VN96] Michael VanHilst and David Notkin. Decoupling change from design. *SIGSOFT Softw. Eng. Notes*, 21:58–69, October 1996.

[Voi09] Janis Voigtländer. Free theorems involving type constructor classes: functional pearl. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 173–184. ACM, 2009.

[Wad89] Philip Wadler. Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, FPCA '89, pages 347–359. ACM, 1989.

[Wad92a] P. Wadler. The essence of functional programming. In *POPL '92*, 1992.

[Wad92b] Philip Wadler. Monads for functional programming. In *Proceedings of the Marktoberdorf Summer School on Program Design Calculi*, August 1992.

[Wad98] P. Wadler. The Expression Problem. Email, November 1998. Discussion on the Java Genericity mailing list.

[WB89] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL '89*, pages 60–76, 1989.