Theorem Proving for Product Lines

Benjamin Delaware William Cook Don Batory

Department of Computer Science
University of Texas at Austin
{bendy,wcook,batory}@cs.utexas.edu

Abstract

Mechanized proof assistants are powerful verification tools, but proof developments can still be difficult and time-consuming. When verifying a family of related programs, the effort can be reduced by proof reuse. In this paper, we show how to engineer proofs for product lines built from feature modules. Each module contains proof fragments which are composed together to build a complete proof of correctness for each product. We consider a product line of programming languages, where each variant includes metatheory proofs verifying the correctness of its syntax and semantic definitions. This approach has been realized in the Coq proof assistant, with the proofs of each feature independently certifiable by Coq. These proofs are composed for each language variant, with Coq mechanically verifying that the composite proofs are correct. As validation, we formalize a core calculus for Java in Coq which can be extended with any combination of casts, interfaces, or generics.

1. Introduction

Mechanized theorem proving is hard: large-scale proof developments [12, 15] take multiple person-years and consist of tens of thousand lines of proof scripts. Given the effort invested in formal verification, it is desirable to reuse as much of the formalization as possible when developing similiar proofs. The problem is compounded when verifying members of a *product line* – a family of related systems [1, 4] – in which the prospect of developing and maintaining individual proofs for each member is untenable.

Product lines can be decomposed into *features* – units of functionality. By selecting and composing different features, members of a product line can be synthesized. The challenge of feature modules is that their contents cut across normal object-oriented boundaries [4, 24]. The same holds for proofs. Feature modularization of proofs is an open, fundamental, and challenging problem.

Surprisingly, the programming language literature is replete with examples of product lines which include proofs. These product lines typically only have two members, consisting of a core language such as *Featherweight Java (FJ)* [13], and an updated one with modified syntax, semantics, and proofs of correctness. Indeed, the original FJ paper also presents *Featherweight Generic Java (FGJ)*, a modified version of FJ with support for generics. An integral part of any type system are the metatheoretic proofs showing *type soundness* – a guarantee that the desired run-time behavior of a language, typically preservation and progress [23], is statically enforced by the type system. As languages become more realistic, the number of features grows, as does the set of possible variants that can be built.

Typically, each research paper only adds a single new feature to a core calculus, and this is accomplished manually. Reuse of existing syntax, semantics, and proofs is achieved by copying existing rules, and in the case of proofs, following the structure of the original proof with appropriate updates. As more features are added, this manual process grows increasingly cumbersome and error prone. Further, the enhanced languages become more difficult to maintain. Adding a feature requires changes that cut across the normal structural boundaries of a language – its syntax, operational semantics, and type system. Each change requires arduously rechecking existing proofs by hand.

Using theorem provers to mechanically formalize languages and their metatheory provides an interesting testbed for studying the modularization of product lines which include proofs. By implementing an extension in the proof assistant as a *feature module*, which includes updates to existing definitions and proofs, we can compose feature modules to build a completely mechanized definition of an enhanced language, with the proofs mechanically checked by the theorem prover. Stepwise development is enabled, and it is possible to start with a core language and add features to progressively build a family or product line of more detailed languages with tool support and less difficulty.

In this paper, we present a methodology for feature-oriented development of a language using a variant of FJ

FJ Expression Syntax		FGJ Expression Syntax
$\mathtt{e} ::= \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.m}\ (\overline{\mathtt{e}}) \mid \mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}}) \mid (\mathtt{C})\ \mathtt{e}$	$\Longmapsto$	$\mathtt{e} ::= \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.m}\ \langle\overline{\mathtt{T}}\rangle^{\beta}\ (\overline{\mathtt{e}}) \mid \mathtt{new}\ \mathtt{C}\ \langle\overline{\mathtt{T}}\rangle^{\beta}\ (\overline{\mathtt{e}}) \mid (\mathtt{C}\ \langle\overline{\mathtt{T}}\rangle^{\beta})\ \mathtt{e}$
FJ Subtyping $\quad \mathtt{T} <: \mathtt{T}$		**FGJ Subtyping** $\quad \Delta^{\delta} \vdash \mathtt{T} <: \mathtt{T}$
		$\Delta \vdash \mathtt{X} <: \Delta(\mathtt{X})$ (GS-VAR)$^{\alpha}$
$\dfrac{\mathtt{S}<:\mathtt{T} \qquad \mathtt{T}<:\mathtt{V}}{\mathtt{S}<:\mathtt{V}}$ (S-TRANS)	$\Longmapsto$	$\dfrac{\Delta^{\delta} \vdash \mathtt{S}<:\mathtt{T} \qquad \Delta^{\delta} \vdash \mathtt{T}<:\mathtt{V}}{\Delta^{\delta} \vdash \mathtt{S}<:\mathtt{V}}$ (GS-TRANS)
$\mathtt{T}<:\mathtt{T}$ (S-REFL)		$\Delta^{\delta} \vdash \mathtt{T}<:\mathtt{T}$ (GS-REFL)
$\dfrac{\mathtt{class}\ \mathtt{C}\ \mathtt{extends}\ \mathtt{D}\ \{\ldots\}}{\mathtt{C}<:\mathtt{D}}$ (S-DIR)		$\dfrac{\mathtt{class}\ \mathtt{C}\ \langle\overline{\mathtt{X} \triangleright \mathtt{N}}\rangle^{\beta}\ \mathtt{extends}\ \mathtt{D}\ \langle\overline{\mathtt{V}}\rangle^{\beta}\ \{\ldots\}}{\Delta^{\delta} \vdash \mathtt{C}\ \langle\overline{\mathtt{T}}\rangle^{\beta} <: [\overline{\mathtt{T}}/\overline{\mathtt{X}}]^{\eta}\ \mathtt{D}\ \langle\overline{\mathtt{V}}\rangle^{\beta}}$ (GS-DIR)
FJ New Typing $\quad \Gamma \vdash \mathtt{e} : \mathtt{T}$		**FGJ New Typing** $\quad \Delta;^{\delta}\Gamma \vdash \mathtt{e} : \mathtt{T}$
$\dfrac{\mathtt{fields}(\mathtt{C}) = \overline{\mathtt{D}}\ \overline{\mathtt{f}} \qquad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{C}} \qquad \overline{\mathtt{C}}<:\overline{\mathtt{D}}}{\Gamma \vdash \mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}}) : \mathtt{C}}$ (T-NEW)	$\Longmapsto$	$\dfrac{\Delta \vdash \mathtt{C}\langle\overline{\mathtt{T}}\rangle^{\gamma} \qquad \mathtt{fields}(\mathtt{C}\ \langle\overline{\mathtt{T}}\rangle^{\beta}) = \overline{\mathtt{V}}\ \overline{\mathtt{f}} \qquad \Delta;^{\delta}\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{U}} \qquad \Delta^{\delta} \vdash \overline{\mathtt{U}}<:\overline{\mathtt{V}}}{\Delta;^{\delta}\Gamma \vdash \mathtt{new}\ \mathtt{C}\ \langle\overline{\mathtt{T}}\rangle^{\beta}\ (\overline{\mathtt{e}}) : \mathtt{C}}$ (GT-NEW)

Figure 1: Selected FJ Definitions with FGJ Changes Highlighted

as a example. We implement feature modules in Coq [7] and demonstrate how to build mechanized proofs that can adapt to new extensions. Each module is a separate Coq file which includes inductive definitions formalizing a language and proofs over those definitions. A feature's proofs can be independently checked by Coq, with no need to recheck existing proofs after composition. We validate our approach through the development of a family of feature-enriched languages, culminating in a generic version of FJ with generic interfaces. Though our work is geared toward mechanized metatheory in Coq, the techniques should apply to different formalizations in other higher-order proof assistants.

2. Background

2.1 On the Importance of Engineering

Architecting product lines (sets of similar programs) has long existed in the software engineering community [17, 22]. So too is achieving object-oriented code reuse in this context [25, 28]. 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 explicitly anticipated in the construction of a feature and its interface. This is no different from using abstract classes and interfaces in the design of OO frameworks [6]. The plug-compatibility of features is not an after-thought but is essential to their design and implementation, allowing the easy integration of new features *as long as* they satisfy the assumptions of existing features. Of course, unanticipated features do arise, requiring a refactoring of existing modules. Again, this is no different than typical software development. Exactly the same ideas hold for modularizing proofs. It is against this backdrop that we motivate our work.

2.2 A Motivating Example

Consider adding generics to the calculus of FJ [13] to produce the FGJ calculus. The required changes are woven throughout the syntax and semantics of FJ. The left-hand column of Figure 1 presents a subset of the syntax of FJ, the rules which formalize the subtyping relation that establish 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.

The categories of changes are tagged in Figure 1 with Greek letters:

FJ Fields of a Supertype Lemma		FGJ Fields of a Supertype Lemma
LEMMA 2.1. *If* $\mathtt{S}<:\mathtt{T}$ *and* $\mathtt{fields(T)} = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$, *then* $\mathtt{fields(S)} = \overline{\mathtt{S}}\ \overline{\mathtt{g}}$ *and* $\mathtt{S_i} = \mathtt{T_i}$ *and* $\mathtt{g_i} = \mathtt{f_i}$ *for all* $\mathtt{i} \leq \#(\mathtt{f})$.		LEMMA 2.2. *If* $\Delta^{\delta} \vdash \mathtt{S}<:\mathtt{T}$ *and* $\mathtt{fields(}\ \mathtt{bound}_{\Delta}\mathtt{(T)))}^{\eta} = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$, *then* $\mathtt{fields(}\ \mathtt{bound}_{\Delta}\mathtt{(S)))}^{\eta} = \overline{\mathtt{S}}\ \overline{\mathtt{g}}$, $\mathtt{S_i} = \mathtt{T_i}$ *and* $\mathtt{g_i} = \mathtt{f_i}$ *for all* $\mathtt{i} \leq \#(\mathtt{f})$.
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 bound.
Case S-REFL $\mathtt{S} = \mathtt{T}$. Follows immediately.		**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, $\mathtt{fields(V)} = \overline{\mathtt{V}}\ \overline{\mathtt{h}}$ and $\mathtt{V_i} = \mathtt{T_i}$ and $\mathtt{h_i} = \mathtt{f_i}$ for all $\mathtt{i} \leq \#(\mathtt{f})$. Again applying the inductive hypothesis, $\mathtt{fields(S)} = \overline{\mathtt{S}}\ \overline{\mathtt{g}}$ and $\mathtt{S_i} = \mathtt{V_i}$ and $\mathtt{g_i} = \mathtt{h_i}$ for all $\mathtt{i} \leq \#(\mathtt{h})$. Since $\#(\mathtt{f}) \leq \#(\mathtt{h})$, the conclusion is immediate.		**Case GS-TRANS** $\Delta^{\delta} \vdash \mathtt{S}<:\mathtt{V}$ and $\Delta^{\delta} \vdash \mathtt{V}<:\mathtt{T}$. By the inductive hypothesis, $\mathtt{fields(}\ \mathtt{bound}_{\Delta}\mathtt{(T)))}^{\eta} = \overline{\mathtt{V}}\ \overline{\mathtt{h}}$ and $\mathtt{V_i} = \mathtt{T_i}$ and $\mathtt{h_i} = \mathtt{f_i}$ for all $\mathtt{i} \leq \#(\mathtt{f})$. Again applying the inductive hypothesis, $\mathtt{fields(}\ \mathtt{bound}_{\Delta}\mathtt{(V)))}^{\eta} = \overline{\mathtt{S}}\ \overline{\mathtt{g}}$ and $\mathtt{S_i} = \mathtt{V_i}$ and $\mathtt{g_i} = \mathtt{h_i}$ for all $\mathtt{i} \leq \#(\mathtt{h})$. Since $\#(\mathtt{f}) \leq \#(\mathtt{h})$, the conclusion is immediate.
Case S-DIR $\mathtt{S} = \mathtt{C}$, $\mathtt{T} = \mathtt{D}$, $\mathtt{class\ C\ extends\ D}\ \{\overline{\mathtt{S}}\ \overline{\mathtt{g}}; \ldots\}$. By the rule F-CLASS, $\mathtt{fields(C)} = \overline{\mathtt{T}}\ \overline{\mathtt{f}}; \overline{\mathtt{S}}\ \overline{\mathtt{g}}$, where $\overline{\mathtt{T}}\ \overline{\mathtt{f}} = \mathtt{fields(D)}$, from which the conclusion is immediate.		**Case GS-DIR** $\mathtt{S} = \mathtt{C}\ \langle\overline{\overline{\mathtt{T}}}\rangle^{\beta}$, $\mathtt{T} = [\overline{\mathtt{T}}/\overline{\mathtt{X}}]^{\eta}\mathtt{D}\ \langle\overline{\mathtt{V}}\rangle^{\beta}$, $\mathtt{class\ C}\langle\overline{\mathtt{X} \triangleleft \mathtt{N}}\rangle\ \mathtt{extends\ D}\langle\overline{\mathtt{V}}\rangle\ \{\overline{\mathtt{S}}\ \overline{\mathtt{g}}; \ldots\}$. By the rule F-CLASS, $\mathtt{fields(C}\ \langle\overline{\overline{\mathtt{T}}}\rangle^{\beta}) = \overline{\mathtt{U}}\ \overline{\mathtt{f}};\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]^{\eta}\overline{\mathtt{S}}\ \overline{\mathtt{g}}$, where $\overline{\mathtt{U}}\ \overline{\mathtt{f}} = \mathtt{fields(}\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]^{\eta}\ \mathtt{D}\ \langle\overline{\overline{\mathtt{V}}}\rangle^{\beta})$. By definition, $\mathtt{bound}_{\Delta}(\mathtt{V}) = \mathtt{V}$ for all non-variable types $\mathtt{V}^{\eta}$, from which the conclusion is immediate.

Figure 2: An Example FJ Proof with FGJ Changes Highlighted

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 to handle this new kind of type.

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

γ. *Adding new premises to existing typing rules to handle modified syntax*. 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 syntax and semantics, the definitions of 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 1, these changes are threaded throughout existing proofs. Consider the related proofs in Figure 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 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 now includes 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. In addition, each different selection of features produces a new language with its own syntax, type system, operational semantics. While the proof of type safety is structurally similar for each language, (potentially subtle) changes occur throughout the proof depending on the features included. By modularizing the type safety proof into distinct features, each language variant is able to build its type safety proof from a common set of proofs. There is no need to manually maintain separate proofs for each language variant. As we shall see, this allows us to add new features to an existing language in a structured way, exploiting existing proofs to build more feature-rich languages.

We demonstrate in the following sections how each kind of extension to a language's syntax and semantics outlined above requires a structural change to a proof. Base proofs can be updated by filling in the pieces required by these changes, enabling reuse of potentially complex proofs for a number of different features. Further, we demonstrate how this modularization can be achieved using the Coq proof assistant. In our approach, each feature has a set of assumptions that serve as extension points, allowing a feature's proofs to be checked independently. As long as an extension provides the necessary proofs to satisfy these assumptions, the composite proof is guaranteed to hold for any composed language. Generating proofs for a composed language is thus a straightforward check that all dependencies are satisfied, with no need to recheck existing proofs.

3. The Structure of Features

Features impose a mathematical structure on the universe of programming languages (including type systems and proofs of correctness) that are to be synthesized. In this section, we review concepts that are essential to our work.

3.1 Features and Feature Compositions

We start with a *base* language or *base* feature to which extensions are added. It is modeled as a constant or zero-ary function. For our study, the *core Featherweight Java* cFJ language is a cast-free variant of FJ. (This omission is not without precedent, as other core calculi for Java [27] omit casts). There are also *optional* features, which are unary functions, that extend the base or other features:

cFJ	core Featherweight Java
Cast	adds casts to expressions
Interface	adds interfaces
Generic	adds type parameters

Assuming no feature interactions, features are composed by function composition. Each expression corresponds to a composite feature or a distinct language. Composing Cast with cFJ builds the original version of FJ:

```
                    cFJ   // Core Featherweight Java
              Cast · cFJ  // Original FJ with Casts [13]
         Interface · cFJ  // Core FJ with Interfaces
  Interface · Cast · cFJ  // Original FJ with Interfaces
           Generic · cFJ  // Core Featherweight Generic Java
    Generic · Cast · cFJ  // Original FGJ
Generic · Interface · cFJ // core Generic FJ with
                          //   Generic Interfaces
     Generic · Interface  // FGJ with
              · Cast · cFJ //   Generic Interfaces
```

3.2 Feature Models

Not all compositions of featues are meaningful. Some features require the presence or absence of other features. An if statement, for example, requires a feature that introduces some notion of booleans to use in test conditions. 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 define a superset of all legal feature expressions, and a set of constraints (the context sensitive part) that eliminates nonsensical sentences [5]. The grammar of feature model P (below) defines eight sentences (features k, i, j are optional; b is mandatory). Contraints limit legal sentences to those that have at least one optional feature, and if feature k is selected, so too must j.

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

Given a sentence of a feature model ('kjb') a dot-product is taken of its terms to map it to an expression ($\mathtt{k} \cdot \mathtt{j} \cdot \mathtt{b}$). A language is synthesized by evaluating the expression. The feature model L that used in our study is context free:

$$\mathtt{L} \ : \ [\mathtt{Generic}]\ [\mathtt{Interface}]\ [\mathtt{Cast}]\ \mathtt{cFJ};$$

3.3 Multiple Representations of Languages

Every base language (cFJ) has multiple representations: its syntax $\mathtt{s}_{cFJ}$, operational semantics $\mathtt{o}_{cFJ}$, type system $\mathtt{t}_{cFJ}$, and metatheory proofs $\mathtt{p}_{cFJ}$. A base language is a tuple of representations $\mathtt{cFJ} = [\mathtt{s}_{cFJ}, \mathtt{o}_{cFJ}, \mathtt{t}_{cFJ}, \mathtt{p}_{cFJ}]$. An optional feature i extends each representation: the language's syntax is extended with new productions $\triangle\mathtt{s_i}$, its operational semantics are extended by modifying existing rules and adding new rules to handle the updated syntax $\triangle\mathtt{o_i}$, etc. Each of these changes is modeled by a unary function. Feature i is a tuple of such functions $\mathtt{i} = [\triangle\mathtt{s_i}, \triangle\mathtt{o_i}, \triangle\mathtt{t_i}, \triangle\mathtt{p_i}]$ that update each representation of a language.

The representations of a language are computed by composing tuples element-wise. The tuple of language $\mathtt{FJ} = \mathtt{Cast} \cdot \mathtt{cFJ}$ is:

$$
\begin{aligned}
\mathtt{FJ} &= \mathtt{Cast} \cdot \mathtt{cFJ} \\
&= [\triangle \mathtt{s}_C, \triangle \mathtt{o}_C, \triangle \mathtt{t}_C, \triangle \mathtt{p}_C] \cdot [\mathtt{s}_{FJ}, \mathtt{o}_{FJ}, \mathtt{t}_{FJ}, \mathtt{p}_{FJ}] \\
&= [\triangle \mathtt{s}_C \cdot \mathtt{s}_{FJ}, \triangle \mathtt{o}_C \cdot \mathtt{o}_{FJ}, \triangle \mathtt{t}_C \cdot \mathtt{t}_{FJ}, \triangle \mathtt{p}_C \cdot \mathtt{p}_{FJ}]
\end{aligned}
$$

That is, the syntax of FJ is the syntax of the base $\mathtt{s}_{FJ}$ composed with extension $\triangle \mathtt{s}_C$, the semantics of FJ are the base semantics $\mathtt{o}_{FJ}$ composed with extension $\triangle \mathtt{o}_C$, and so on. In this way, all parts of a language are updated lock-step when features are composed. See [4, 11] for generalizations of these ideas.

3.4 Feature Interactions

Feature interactions are ubiquitous. Consider the Interface feature which introduces syntax for interface declarations:

$$\mathtt{J ::= interface\ I}\ \{\overline{\mathtt{Mty}}\}$$

This declaration may be changed by other features. When Generic is added, the syntax of an interface declaration must be updated to include type parameters:

$$\mathtt{J ::= interface\ I}\ \boxed{\langle \overline{\mathtt{X} \triangleright \mathtt{N}} \rangle}\ \{\overline{\mathtt{Mty}}\}$$

Similarly, any proofs in Generic that induct over the derivation of the subtyping judgement must add new cases for the subtyping rule introduced by the Interface feature. Such proof updates are necessary only when *both* features are present. The set of changes made across all representations is the *interaction* of these features, written Generic#Interface.[1]

Until now, features were composed by only one operation (dot or $\cdot$). Now we introduce two additional operations: product ($\times$) and interaction ($\#$). When designers want a set of features, they really want the $\times$-product of these features, which includes the dot-product of these features *and* their interactions. The $\times$-product of features f and g is:

$$\mathtt{f} \times \mathtt{g} \;=\; (\mathtt{f}\#\mathtt{g}) \cdot \mathtt{f} \cdot \mathtt{g} \qquad (1)$$

where $\#$ distributes over dot and $\#$ takes precedence over dot:

$$\mathtt{f}\#(\mathtt{g} \cdot \mathtt{h}) \;=\; (\mathtt{f}\#\mathtt{g}) \cdot (\mathtt{f}\#\mathtt{h}) \qquad (2)$$

[1] Our Generic#Interface example is isomorphic to the classical example of fire and flood control [14]. 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. flood · b and fire · b) is straightforward. However, adding both (flood · fire · 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 flood#fire, which is the interaction of flood and fire. flood#fire *represents the changes (extension) that is needed to make the* flood *and* fire *features work correctly together*. The correct building design is flood#fire · flood · fire · b.

That is, the interaction of a feature with a dot-product is the dot-product of their interactions. $\times$ is right-associative and $\#$ is associative and commutative.[2]

The connection of $\times$ and $\#$ to prior discussions is simple. A sentence of a feature model ('kjb') is mapped to an expression by a $\times$-product of its terms ($\mathtt{k} \times \mathtt{j} \times \mathtt{b}$). Equations (1) and (2) are used to reduce an expression with $\times$ operations to an expression with only dot and $\#$, as below:

$$
\begin{aligned}
\mathtt{p} &= \mathtt{k} \times \mathtt{j} \times \mathtt{b} && \text{// def of p} \\
&= \mathtt{k} \times (\, \mathtt{j}\#\mathtt{b} \cdot \mathtt{j} \cdot \mathtt{b}\,) && \text{// (1)} \\
&= \mathtt{k}\#(\mathtt{j}\#\mathtt{b} \cdot \mathtt{j} \cdot \mathtt{b}) \;\cdot\; \mathtt{k} \cdot\; (\mathtt{j}\#\mathtt{b} \cdot \mathtt{j} \cdot \mathtt{b}) && \text{// (1)} \\
&= \mathtt{k}\#\mathtt{j}\#\mathtt{b} \cdot \mathtt{k}\#\mathtt{j} \cdot \mathtt{k}\#\mathtt{b} \cdot \mathtt{k} \cdot \mathtt{j}\#\mathtt{b} \cdot \mathtt{j} \cdot \mathtt{b} && \text{// (2)} \qquad (4)
\end{aligned}
$$

Language p is synthesized by evaluating expression (4). Interpreting modules for individual features like k, j, and b as 1-way feature interactions (where k#j denotes a 2-way interaction and k#j#b is 3-way), the universe of modules in a feature-oriented construction are exclusively those of feature interactions.

An $\times$-product of n features results in $2^{\mathtt{n}}$ interactions (i.e. all possible feature combinations). Fortunately, the *vast* majority of feature interactions are empty, meaning that they correspond to the identity transformation 1, whose properties are:

$$\mathtt{1} \cdot \mathtt{f} \;=\; \mathtt{f} \cdot \mathtt{1} \;=\; \mathtt{f} \qquad (3)$$

Most non-empty interactions are pairwise (2-way). Occasionally higher-order interactions arise. The $\times$-product of cFJ, Interface, and Generic is:

$$
\begin{aligned}
&\mathtt{Generic} \times \mathtt{Interface} \times \mathtt{cFJ} \\
&\quad = \mathtt{Generic\#Interface\#cFJ} \cdot \mathtt{Generic\#Interface} \\
&\qquad \cdot \mathtt{Generic\#cFJ} \cdot\; \mathtt{Generic} \cdot \mathtt{Interface\#cFJ} \\
&\qquad \cdot \mathtt{Interface} \cdot \mathtt{cFJ} \\
&\quad = \mathtt{Generic\#Interface} \cdot \mathtt{Generic} \cdot \mathtt{Interface} \cdot \mathtt{cFJ}
\end{aligned}
$$

which means that all 2- and 3-way interactions, except Generic#Interface, equal 1. In our case study, the complete set of interaction modules that are not equal to 1 is:

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

Each of these interaction modules is represented by a tuple of definitions or a tuple of changes to these definitions.

[2] A more general algebra has operations $\times$, $\#$, and $\cdot$ that are all associative and commutative [3]. This generality is not needed for this paper.

4. Our Approach

We design features to be monotonic: what was true before a feature is added remains valid after composition, although the scope of validity may be qualified. This is standard in feature-based designs, as it simplifies reasoning with features [1].

All representations of a language (syntax, operational semantics, type system, proofs) are written in distinct languages. Language syntax uses BNF, operational semantics and type systems use standard rule notations, and metatheoretic proofs are formal proofs in Coq.

Despite these different representations, there are only two kinds of changes that a feature makes to a document: new definitions can be added and existing definitions can be modified. Addition is just the union of definitions. Modification requires definitions to be engineered for change.

In the following sections, we explain how to accomplish addition and modification. We alert readers that our techniques for extending language syntax are identical to extension techniques for the other representations. The critical contribution of our approach is how we guarantee the correctness of composed proofs, the topic of Section 4.5.

4.1 Language Syntax

We use BNF to express language syntax. Figure 3a shows the BNF for expressions in `cFJ`, Figure 3b the production that the `Cast` feature adds to `cFJ`'s BNF, and Figure 3c the composition (union) of these productions, that defines the expression grammar of the `FJ` = `Cast` · `cFJ` language (Figure 1).

$$\begin{array}{lcl} \texttt{E} & : & (\texttt{C})\,\texttt{E}\,; \end{array}$$

(b)

$$\begin{array}{lcl} \texttt{E} & : & \texttt{x} \\ & | & \texttt{E.f} \\ & | & \texttt{E.m}(\overline{\texttt{E}}) \\ & | & \texttt{new C}(\overline{\texttt{E}})\ ; \end{array}$$

(a)

$$\begin{array}{lcl} \texttt{E} & : & \texttt{x} \\ & | & \texttt{E.f} \\ & | & \texttt{E.m}(\overline{\texttt{E}}) \\ & | & \texttt{new C}(\overline{\texttt{E}}) \\ & | & (\texttt{C})\,\texttt{E}\,; \end{array}$$

(c)

Figure 3: Union of Grammars

Modifying existing productions requires foresight to anticipate how productions may be changed by other features. (This is no different from object-oriented refactorings that prepare source code for extensions – visitors, frameworks, strategies, etc. – as discussed in Section 2.) Consider the impact of adding the `Generics` feature to `cFJ` and `Cast`: type parameters must be added to the expression syntax of method calls and class types now have type parameters. What we do is to insert *variation points (VP)*, a standard concept in product line designs, to allow new syntax to appear in a production. For syntax rules, a VP is simply the name of an (initially) empty production.

Figure 4a-b shows the VPs $\texttt{TP}_\texttt{m}$ added to method calls in the `cFJ` expression grammar and $\texttt{TP}_\texttt{t}$ added to class types in the `cFJ` and `Cast` expression grammars. Figure 4c shows the composition (union) of the revised `Cast` and `cFJ` expression grammars. Since $\texttt{TP}_\texttt{m}$ and $\texttt{TP}_\texttt{t}$ are empty, Figure 4c can be simplified to the grammar in Figure 3c.

$$\begin{array}{lcl} \texttt{E} & : & \texttt{x} \\ & | & \texttt{E.f} \\ & | & \boxed{\texttt{TP}_\texttt{m}}\ \texttt{E.m}\ (\overline{\texttt{E}}) \\ & | & \texttt{new}\ (\ \boxed{\texttt{TP}_\texttt{t}}\ \texttt{C})\ (\overline{\texttt{E}}); \\ \texttt{TP}_\texttt{m} & : & \epsilon; \\ \texttt{TP}_\texttt{t} & : & \epsilon; \end{array}$$

(a)

$$\begin{array}{lcl} \texttt{E} & : & (\ \boxed{\texttt{TP}_\texttt{t}}\ \texttt{C})\ \texttt{E}; \end{array}$$

(b)

$$\begin{array}{lcl} \texttt{E} & : & \texttt{x} \\ & | & \texttt{E.f} \\ & | & \boxed{\texttt{TP}_\texttt{m}}\ \texttt{E.m}\ (\overline{\texttt{E}}) \\ & | & \texttt{new}\ (\ \boxed{\texttt{TP}_\texttt{t}}\ \texttt{C})\ (\overline{\texttt{E}}) \\ & | & (\ \boxed{\texttt{TP}_\texttt{t}}\ \texttt{C})\ \texttt{E}; \\ \texttt{TP}_\texttt{m} & : & \epsilon; \\ \texttt{TP}_\texttt{t} & : & \epsilon; \end{array}$$

(c)

Figure 4: Modification of Grammars

Now consider the changes that `Generics` makes to expression syntax: it redefines $\texttt{TP}_\texttt{m}$ and $\texttt{TP}_\texttt{t}$ to be lists of type parameters, thereby updating all productions that reference these VPs. Figure 5a shows this definition. Figure 5b shows the productions of Figure 4c with these productions inlined, building the expression grammar for `Generic` · `Cast` · `cFJ`.

$$\begin{array}{lcl} \texttt{TP}_\texttt{m} & : & \langle\overline{\texttt{T}}\rangle; \\ \texttt{TP}_\texttt{t} & : & \langle\overline{\texttt{T}}\rangle; \end{array}$$

(a)

$$\begin{array}{lcl} \texttt{E} & : & \texttt{x} \\ & | & \texttt{E.f} \\ & | & \langle\overline{\texttt{T}}\rangle\ \texttt{E.m}\ (\overline{\texttt{E}}) \\ & | & \texttt{new}\ (\langle\overline{\texttt{T}}\rangle\ \texttt{C})\ \ (\overline{\texttt{E}}) \\ & | & (\langle\overline{\texttt{T}}\rangle\ \texttt{C})\ \texttt{E}\ ; \end{array}$$

(b)

Figure 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 [18]). Framework hook methods are initially empty and users can override them with a definition that is specific to their context. We do the same here.

These are simple and intuitively appealing techniques for defining and composing language extensions. As readers will see, these same ideas apply to rules and proofs as well.

4.2 Reduction and Typing Rules

The judgments that form the operational semantics and type system of a language are defined by rules. Figure 6a shows the typing rules for `cFJ` expressions, Figure 6b the rule that the `Cast` feature adds, and Figure 6c the composition (union) of these rules, defining the typing rules for `FJ`.

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

$$\frac{\texttt{fields}(\texttt{C}) = \overline{\texttt{V}}\ \overline{\texttt{f}} \qquad \Gamma \vdash \overline{\texttt{e}} : \overline{\texttt{U}} \qquad \overline{\texttt{U}} <: \overline{\texttt{V}}}{\Gamma \vdash \texttt{new C}(\overline{\texttt{e}}) : \texttt{C}} \ (\text{T-New})$$

$\vdots$

(a)

$$\frac{\Gamma \vdash \texttt{e}_0 : \texttt{D} \qquad \texttt{D} <: \texttt{C}}{\Gamma \vdash \texttt{e}_0 : \texttt{C}} \ (\text{T-UCast})$$

(b)

$$\frac{\texttt{fields}(\texttt{C}) = \overline{\texttt{V}}\ \overline{\texttt{f}} \qquad \Gamma \vdash \overline{\texttt{e}} : \overline{\texttt{U}} \qquad \overline{\texttt{U}} <: \overline{\texttt{V}}}{\Gamma \vdash \texttt{new C}(\overline{\texttt{e}}) : \texttt{C}} \ (\text{T-New})$$

$\vdots$

$$\frac{\Gamma \vdash \texttt{e}_0 : \texttt{D} \qquad \texttt{D} <: \texttt{C}}{\Gamma \vdash \texttt{e}_0 : \texttt{C}} \ (\text{T-UCast})$$

(c)

Figure 6: Union of Typing Rules

extend a judgement's signature, and (c) functions that transform existing premises and conclusions. Predicate and relational holes are empty by default. 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 $\texttt{WF}_\texttt{c}(\texttt{D}, \texttt{TP}_\texttt{t}\ \texttt{C})$ predicate and for the D relational hole in the cFJ expression grammar. (Compare Figure 6a to its extension in Figure 7a). Figure 7b shows the non-empty definitions for these VPs introduced by the Generic feature, with Figure 7c showing the T-New rule with these definitions inlined.

$$\frac{\texttt{WF}_\texttt{c}(\texttt{D}, \texttt{TP}_\texttt{t}\ \texttt{C}) \qquad \texttt{fields}(\texttt{TP}_\texttt{t}\ \texttt{C}) = \overline{\texttt{V}}\ \overline{\texttt{f}} \qquad \texttt{D}; \Gamma \vdash \overline{\texttt{e}} : \overline{\texttt{U}} \qquad \texttt{D} \vdash \overline{\texttt{U}} <: \overline{\texttt{V}}}{\texttt{D}; \Gamma \vdash \texttt{new}(\texttt{TP}_\texttt{t}\ \texttt{C})(\overline{\texttt{e}}) : \texttt{TP}_\texttt{t}\ \texttt{C}} \ (\text{T-New})$$

$$\frac{\mathbb{T}}{\texttt{WF}_\texttt{c}(\epsilon, \texttt{C}, \epsilon)}$$

$\texttt{D} := \epsilon$

$\vdots$

(a)

$$\frac{\Delta \vdash \langle\overline{\texttt{T}}\rangle\texttt{C ok}}{\texttt{WF}_\texttt{c}(\Delta, \langle\overline{\texttt{T}}\rangle\ \texttt{C})}$$

$\texttt{D} := \Delta$

(b)

$$\frac{\Delta \vdash \langle\overline{\texttt{T}}\rangle\texttt{C ok} \qquad \texttt{fields}(\langle\overline{\texttt{T}}\rangle\texttt{C}) = \overline{\texttt{V}}\ \overline{\texttt{f}} \qquad \Delta; \Gamma \vdash \overline{\texttt{e}} : \overline{\texttt{U}} \qquad \Delta \vdash \overline{\texttt{U}} <: \overline{\texttt{V}}}{\Delta; \Gamma \vdash \texttt{new}(\langle\overline{\texttt{T}}\rangle\texttt{C})(\overline{\texttt{e}}) : \langle\overline{\texttt{T}}\rangle\texttt{C}} \ (\text{GT-New})$$

$\vdots$

(c)

Figure 7: Building Generic Typing Rules

4.3 Implementing Feature Modules in Coq

The syntax, operational semantics, and typing rules of a language are embedded in Coq as standard inductive data types. The metatheoretic proofs of a language are then written over these encodings. Figure 8a-b gives the Coq definitions for the syntax of Figure 3a and the typing rules of Figure 7a. A feature module in Coq is realized as a Coq file containing its definitions and proofs. The target language is itself a Coq file which combines the definitions and proofs from a set of Coq feature modules. The appendix includes more details on feature composition in Coq.

```
Definition TP_m := unit.
Definition TP_t := unit.
Inductive C : Set :=
  | ty : TP_t → Name → E.
Inductive E : Set :=
  | e_var : Var → E
  | fd_access : E → F → E
  | m_call : TP_m → E → M → List E → E
  | new : C → List E → E.
```

(a)

```
Definition Context := Var_Context.
Definition WF_c (gamma : Context)(c : C):= True.
Inductive Exp_WF : Context → E → Ty → Prop :=
  | T_New : forall gamma c us tp d_fds es,
     WF_c gamma (ty tp c) →
     fields (ty tp c) d_fds →
     Exps_WF gamma es us →
     subtypes gamma us d_fds →
     Exp_WF gamma (new (ty tp c) es) (ty tp c).
     ⋮
```

(b)

Figure 8: Coq Encoding of Fig. 3a and Fig. 7a.

As shown in Figure 8, each feature includes the default definitions for its variation points. When composed with features that provide new definitions for a variation point, these definitions are updated for the target language. In the case of syntax, the final definition of a VP is the juxtaposition of the definitions from each of the features. For abstract predicates, the target predicate is the conjuction of all the VPs. The Coq encoding of expressions the Cast, and Generic features and the result of their composition with cFJ is given in Figure 9.

OO frameworks are implemented using inheritance and mixin layers [2], techniques that are not available in most proof assistants. To support case extension and VPs, our Coq feature modules rely on the parameterization mechanisms of the Coq theorem prover. To take the union of syntax, reduction and typing rules, feature modules are parameterized by the final set of rules. This parameter is used for subterms, leaving the recursion in inductive definitions open. This allows a feature's definitions to use any term from the final language as a subterm. To close the inductive loop, the target language instantiates this parameter with the data type defining the target syntax. Similarly, the VPs in each module are explicitly represented as abstract sets/predicates/functions that are instantiated by the final language. See the Appendix for a complete example.

```
Inductive E : Set :=
  | cast : C → E → E.
```

Cast

```
Definition TP_m := list Type.
Definition TP_t := list Type.
```

Generic

```
Definition TP_m := (list Type, unit).
Definition TP_t := (list Type, unit).
Inductive C : Set :=
  | ty : TP_t → Name → E.
Inductive E : Set :=
  | e_var : Var → E
  | fd_access : E → F → E
  | m_call : TP_m → E → M → List E → E
  | new : C → List E → E
  | cast : C → E → E.
```

cFJ · Cast · Generic

Figure 9: Coq Encoding of Fig. 3a and Fig. 6a.

4.4 Theorem Statements

Variation points can appear in the statements of lemmas and theorems, enabling the construction of extensible proofs. Consider the lemma in Figure 10 with its seven VPs.

$\mathtt{TP_t}$: VP for Class Types
$\mathtt{TP_m}$: VP for Method Call Expression
μ : VP for Method Types
$\mathtt{D}$: Relational Hole for Typing Rules
$WF_{mc}(\mathtt{D}, \mu, \mathtt{TP_m})$: Predicate for T-INVK
$WF_{ne}(\mathtt{D}, \mathtt{TP_t}\ \mathtt{C})$: Predicate for T-NEW
$\Phi_{\mathtt{M}}(\mathtt{TP_m}, \mu, \overline{\mathtt{T}})$: Transformation for Return Types

LEMMA 4.1 (Well-Formed MBody). *If* $\mathtt{mtype}(\mathtt{m}, \mathtt{TP_t}\ \mathtt{C}) = \mu\ \overline{\mathtt{V}} \rightarrow \mathtt{V}$, *and* $WF_{mc}(\mathtt{D}, \mu, \mathtt{TP_m})$ *with* $\mathtt{mbody}(\mathtt{TP_1}, \mathtt{m}, \mathtt{TP_t}\ \mathtt{C}) = \overline{\mathtt{x}}.\mathtt{e}$, *where* $WF_{ne}(\mathtt{D}, \mathtt{TP_t}\ \mathtt{C})$, *then there exists some* $\mathtt{N}$ *and* $\mathtt{S}$ *such that* $\mathtt{D} \vdash \mathtt{TP_t}\ \mathtt{C} <: \mathtt{N}$ *and* $\mathtt{D} \vdash \mathtt{S} <: \Phi_{\mathtt{M}}(\mathtt{TP_m}, \mu, \mathtt{V})$ *and* $\mathtt{D}; \overline{\mathtt{x}} : \Phi_{\mathtt{M}}(\mathtt{TP_m}, \mu, \overline{\mathtt{V}}), \mathtt{this} : \mathtt{N} \vdash \mathtt{e} : \mathtt{S}$.

Figure 10: VPs in a Parameterized Lemma Statement

Different instantiations of VPs produce variations of the original productions and rules, with the lemma adapting accordingly. Figure 11 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. Predicates that are always true are dropped from the definition.

Without an accompanying proof, extensible theorem statements are uninteresting. Ideally, a proof should adapt to any VP instantiation or case introduction, allowing the proof to be reused in any target language variant. Of course, proofs must rule out broken extensions which do not guarantee progress and preservation, and admit only "correct" new cases or VP instantiations. This is the key challenge in crafting modular proofs.

4.5 Crafting Modular Proofs

Rather than writing multiple related proofs, we want to create a single proof for a generic statement of a theorem. The proof is then specialized for the target language by instantiating the variation points appropriately. Instead of separately proving the two lemmas in Figure 2, the cFJ feature has a single proof of the generic Lemma 4.4 (Figure 12). This lemma is specialized to the variants FJ and FGJ shown in Figure 2. The proof now reasons over the generic subtyping rules with variation points, as in the case for **S-Dir** in Figure 12. From the (human or computer) theorem prover's point of view, these holes are opaque. Thus, this proof becomes stuck when it requires knowledge about behavior of $\Phi_{\mathtt{f}}$.

LEMMA 4.4. *If* $\Delta \vdash \mathtt{S} <: \mathtt{T}$ *and* $\mathtt{fields}(\Phi_{\mathtt{f}}(\Delta, \mathtt{T})) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$, *then* $\mathtt{fields}(\Phi_{\mathtt{f}}(\Delta, \mathtt{S})) = \overline{\mathtt{S}}\ \overline{\mathtt{g}}$, $\mathtt{S_i} = \mathtt{T_i}$ *and* $\mathtt{g_i} = \mathtt{f_i}$ *for all* $\mathtt{i} \leq \#(\mathtt{f})$.

Case S-DIR
$\mathtt{S} = \mathtt{TP_0}\ \mathtt{C}, \mathtt{CP_0}\ \mathtt{class}\ \mathtt{C}\ \mathtt{extends}\ \mathtt{TP_1}\ \mathtt{D}\ \{\overline{\mathtt{S}}\ \overline{\mathtt{g}}; \ldots\}$,
$\mathtt{T} = \Phi_{SD}(\mathtt{TP_0}, \mathtt{CP_0}, \mathtt{TP_1}\ \mathtt{D})$.
By the rule F-CLASS,
$\mathtt{fields}(\mathtt{TP_0}\ \mathtt{C}) = \overline{\mathtt{U}}\ \overline{\mathtt{f}}; \Phi_{SD}(\mathtt{TP_0}, \mathtt{CP_0}, \overline{\mathtt{S}})\ \overline{\mathtt{g}}$, where $\mathtt{fields}(\Phi_{SD}(\mathtt{TP_0}, \mathtt{CP_0}, \mathtt{TP_1}\ \mathtt{D})) = \overline{\mathtt{U}}\ \overline{\mathtt{f}}$. Assuming that for all class types $\mathtt{TP_2}\ \mathtt{D}'$, $\Phi_{\mathtt{f}}(\Delta, \mathtt{TP_2}\ \mathtt{D}') = \mathtt{TP_2}\ \mathtt{D}'$ and $\Phi_{SD}(\mathtt{TP_0}, \mathtt{CP_0}, \mathtt{TP_2}\ \mathtt{D}')$ returns a class type, the conclusion is immediate.

Figure 12: Generic Statement of Lemmas 2.2 and 2.1 and proof for **S-Dir** case.

In order to proceed, the lemma must constrain possible VP instantiations, namely that they have the properties required by the proof. In the case of Lemma 4.4, this behavior is that $\Phi_{\mathtt{f}}$ must be the identity function for non-variable types and that Φ_{SD} maps class types to class types. In order for this proof to hold for the target language, the instantiations of $\Phi_{\mathtt{f}}$ and Φ_{SD} must have this property. More concretely, the proof assumes this behavior for all instantiations of $\Phi_{\mathtt{f}}$ and Φ_SD, producing the new generic Lemma 4.5. In order to produce the desired lemma, the target language instantiates the VPs and provides proofs of all the assumed behaviors. Each feature which supplies a concrete realization of a VP also provides the necessary proofs about its behavior. The assumptions of a proof form an explicit interface against which a proof is written. The union of all assumptions define the interface for the feature – as long as the target language satisfies this interface, a feature's generic proofs can be specialized and reused in their entirety.

We also have to deal with new cases. Whenever a new rule or production is added, a new case must be added to proofs

$\mathtt{TP_t} : \epsilon; \quad \mathtt{TP_m} : \epsilon; \quad \mu : \epsilon;$ $\mathtt{D} := \epsilon \qquad \dfrac{\mathbb{T}}{WF_{mc}(\epsilon, \epsilon, \overline{\mathtt{T}})}$ $\dfrac{\mathbb{T}}{WF_{ne}(\epsilon, \mathtt{C})} \qquad \Phi_{\mathtt{M}}(\epsilon, \epsilon, \overline{\mathtt{T}}) := \overline{\mathtt{T}}$	LEMMA 4.2 (Well-Formed MBody). *If* $\mathtt{mtype}(\mathtt{m}, \mathtt{C_0}) = \overline{\mathtt{V}} \rightarrow \mathtt{V}$ *and* $\mathtt{mbody}(\mathtt{m}, \mathtt{V_0}) = \overline{\mathtt{x}}.\mathtt{e}$*, then there exists* N *and* S *such that* $\vdash \mathtt{C_0} <: \mathtt{N}$ *and* $\vdash \mathtt{S} <: \mathtt{V}$ *and* $\overline{\mathtt{x}} : \overline{\mathtt{V}}, \mathtt{this} : \mathtt{N} \vdash \mathtt{e} : \mathtt{S}$.
$\mathtt{TP_t} : \overline{\mathtt{T}}; \quad \mathtt{TP_m} : \overline{\mathtt{T}}; \quad \mu : \langle\overline{\mathtt{Y} \triangleright \mathtt{P}}\rangle;$ $\mathtt{D} := \Delta \qquad \dfrac{\Delta \vdash \overline{\mathtt{V}}\ \mathtt{ok} \qquad \Delta \vdash \overline{\mathtt{V}} <: [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}}}{WF_{mc}(\Delta, \langle\overline{\mathtt{Y} \triangleright \mathtt{P}}\rangle, \overline{\mathtt{V}})}$ $\dfrac{\Delta \vdash \langle\overline{\mathtt{T}}\rangle\mathtt{C}\ \mathtt{ok}}{WF_{ne}(\Delta, \langle\overline{\mathtt{T}}\rangle\mathtt{C})} \qquad \Phi_{\mathtt{M}}(\langle\overline{\mathtt{T}}\rangle, \langle\overline{\mathtt{Y} \triangleright \mathtt{P}}\rangle, \overline{\mathtt{V}}) := [\overline{\mathtt{T}}/\overline{\mathtt{Y}}]\overline{\mathtt{V}}$	LEMMA 4.3 (Well-Formed MBody). *If* $\mathtt{mtype}(\mathtt{m}, \langle\overline{\mathtt{T}}\rangle\mathtt{C}) = \langle\overline{\mathtt{Y} \triangleright \mathtt{P}}\rangle\overline{\mathtt{U}} \rightarrow \mathtt{U}$ *and* $\Delta \vdash \langle\overline{\mathtt{T}}\rangle\mathtt{C}\ \mathtt{ok}$ *with* $\mathtt{mbody}(\langle\overline{\mathtt{V}}\rangle, \mathtt{m}, \langle\overline{\mathtt{T}}\rangle\mathtt{C}) = \overline{\mathtt{x}}.\mathtt{e}$*, where* $\Delta \vdash \overline{\mathtt{V}}\ \mathtt{ok}$ *and* $\Delta \vdash \overline{\mathtt{V}} <: [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}}$*, then there exists some* N *and* S *such that* $\Delta \vdash \langle\overline{\mathtt{T}}\rangle\mathtt{C} <: \mathtt{N}$ *and* $\Delta \vdash \mathtt{S} <: [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}$ *and* $\Delta; \overline{\mathtt{x}} : [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}, \mathtt{this} : \mathtt{N} \vdash \mathtt{e} : \mathtt{S}$

Figure 11: Statements of Lemma 4.1 for cFJ and FGJ

LEMMA 4.5. *As long as* $\Phi_{\mathtt{f}}(\Delta, \mathtt{V}) = \mathtt{V}$ *for all non-variable types* V *and* Φ_{SD} *maps class types to class types*, *if* $\Delta \vdash \mathtt{S} <: \mathtt{T}$ *and* $\mathtt{fields}(\Phi_{\mathtt{f}}(\Delta, \mathtt{T})) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$*, then* $\mathtt{fields}(\Phi_{\mathtt{f}}(\Delta, \mathtt{S})) = \overline{\mathtt{S}}\ \overline{\mathtt{g}}$*,* $\mathtt{S_i} = \mathtt{T_i}$ *and* $\mathtt{g_i} = \mathtt{f_i}$ *for all* $\mathtt{i} \leq \#(\mathtt{f})$.

which induct over or case split on the original production or rule. For FGJ, this means that a new case must be added for **GS-Var**. When writing an inductive proof, a feature provides cases for each of the rules or productions it introduces. To build the proof for the target language, a new skeleton proof by induction is started. Each of the cases is discharged by the proof given in the introducing feature.

4.6 Engineering Extensible Proofs in Coq

Each Coq feature module contains proofs for the extensible lemmas it provides. To get a handle on the behavior of opaque parameters, Coq feature modules make explicit assumptions about their behavior. Just as definitions were parameterized on extension points, proofs are now parameterized on a set of lemmas that define legal extensions. These assumptions enable separate certification of feature modules. Coq certifies that a proof is correct for all instantiations or case introductions that satisfy its assumptions, enabling proof reuse for all compatible features.

As a concrete example, consider the Coq proof of Lemma 4.6 given in Figure 13. The cFJ feature provides the statement of the lemma, which is over the abstract subtype relation. Both the Generic and cFJ features give proofs for their definitions of the subtype relation. Notably, the Generic feature assumes that if a type variable is found in a Context Gamma, it will have the same value in app_context Gamma Delta for all Contexts Delta. Any compatible extension of Context and app_Context can thus reuse this proof.

To build the final proof, the target language inducts over subtype, as shown in the final box of Figure 13. For each constructor, the lemma dispatches to the proofs from the corresponding feature module. To reuse those proofs, each

```
Variables (app_context : Context → Context → Context)
(FJ_subtype_Wrap : forall gamma S T,
  FJ_subtype gamma S T → subtype gamma S T).
Definition Weaken_Subtype_app_P
  delta S T (sub_S_T : subtype delta S T) :=
  forall gamma, subtype (app_context delta gamma) S T.

Lemma cFJ_Weaken_Subtype_app_H1 :
  forall (ty : Ty) (gamma : Context),
  Weaken_Subtype_app_P _ _ _ (sub_refl ty gamma).
Lemma cFJ_Weaken_Subtype_app_H2 : forall c d e
  gamma sub_c sub_d,
  Weaken_Subtype_app_P _ _ _ sub_c →
  Weaken_Subtype_app_P _ _ _ sub_d →
  Weaken_Subtype_app_P _ _ _ (sub_trans c d e gamma sub_c sub_d).
Lemma cFJ_Weaken_Subtype_app_H3 :
  forall ce c d fs k' ms te te' delta CT_c
  bld_te, Weaken_Subtype_app_P _ _ _
   (sub_dir ce c d fs
     k' ms te te' delta CT_c bld_te).
Definition cFJ_Weaken_Subtype_app :=
  cFJ_subtype_ind _ cFJ_Weaken_Subtype_app_H1
  cFJ_Weaken_Subtype_app_H2 cFJ_Weaken_Subtype_app_H3.
```

```
Variables (app_context:Context → Context → Context)
  (TLookup_app : forall gamma delta X ty,
      TLookup gamma X ty →
      TLookup (app_context gamma delta) X ty).
  (GJ_subtype_Wrap : forall gamma S T,
    GJ_subtype gamma S T → subtype gamma S T).
Definition Weaken_Subtype_app_P :=
  cFJ_Pinitions.Weaken_Subtype_app_P _ _ subtype app_context.

Lemma GJ_Weaken_Subtype_app : forall gamma
 S T (sub_S_T : GJ_subtype gamma S T),
 Weaken_Subtype_app_P _ _ _ sub_S_T.
  cbv beta delta; intros; apply GJ_subtype_Wrap.
  inversion sub_S_T; subst.
  econstructor; eapply TLookup_app; eauto.
Qed.
```

```
Fixpoint Weaken_Subtype_app gamma S T
 (sub_S_T : subtype gamma S T) :
 Weaken_Subtype_app_P _ _ _ sub_S_T :=
  match sub_S_T return Weaken_Subtype_app_P _ _ _ sub_S_T with
  | cFJ_subtype_Wrap gamma S' T' sub_S_T' ⇒
    cFJ_Weaken_Subtype_app _ _ _ _ _ _ cFJ_Ty_Wrap _ _ _ CT
      _ subtype GJ_Phi_sb cFJ_subtype_Wrap app_context _ _ _
      sub_S_T' Weaken_Subtype_app
  | GJ_subtype_Wrap gamma S' T' sub_S_T' ⇒
    GJ_Weaken_Subtype_app _ _ Gty _ TLookup subtype
    GJ_subtype_Wrap app_context TLookup_app' _ _ _ sub_S_T'
    end.
```

Figure 13: Coq proofs of Lemma 4.6 for the cFJ and Generic features and the composite proof.

LEMMA 4.6 (Subtype Weakening). *For all contexts* Γ *and* Δ, *if* $\Gamma \vdash \texttt{S<:T}$, $\Gamma; \Delta \vdash \texttt{S<:T}$.

Figure 14: Weakening lemma for subtyping.

of their assumptions has to be satisfied by a theorem (e.g. `TLookup_app'` is provided to `TLookup_app`). The inductive hypothesis is provided to `cFJ_Weaken_subtype_app` for use on its subterms. As long as every assumption is satisfied for each proof case, Coq will certify the composite proof. There is one important caveat: proofs which use the inductive hypothesis can only do so on subterms or subjudgements. By using custom induction schemes to build proofs, features can ensure that this check will always succeed. The `cFJ_subtype_ind` induction scheme used to combine `cFJ`'s cases in the first box of Figure 13 is an example.

5. Implementation

We implemented the six feature modules of Section 3.4 in the Coq proof assistant. Each contains pieces of syntax, semantics, type system, and metatheoretic proofs needed by that feature or interaction. Using them, we built the seven variants on Featherweight Java listed in Section 3.2[3].

Module	Lines of Code in Coq
`cFJ`	2582 LOC
`Cast`	439 LOC
`Interface`	450 LOC
`Generic`	4924 LOC
`Generic#Interfaces`	1360 LOC
`Generic#Cast`	265 LOC

Figure 15: Feature Module Sizes

While we achieve feature composition by manually instantiating these modules, the process is straightforward and should be mechanized. Except for some trivial lemmas, the proofs for the final language are assembled from proof pieces from its constituent features by supplying them with lemmas which satisfy their assumptions. Importantly, once the proofs in each of the feature modules have been certified by Coq, they do not need to be rechecked for the target language. Any proof is guaranteed to be correct for any language which satisfies the interface formed by the set of assumptions for that lemma. This has a practical effect as well: certifying larger feature modules takes a non-trivial amount of time. Figure 16 lists the certification times for the feature modules in Figure 15 and the language variants built from their composition. By checking the proofs of each feature in isolation, Coq is able to certify the entire product line in roughly the same amount of time as the `cFJ` feature module. Rechecking the work of each feature for each individual product would quickly become expensive. Independent certification is particularly useful when modifying a single feature. Recertifying the product line is a matter of rechecking the proofs of the modified features and then performing a quick check of the products, without having to recheck the independent features.

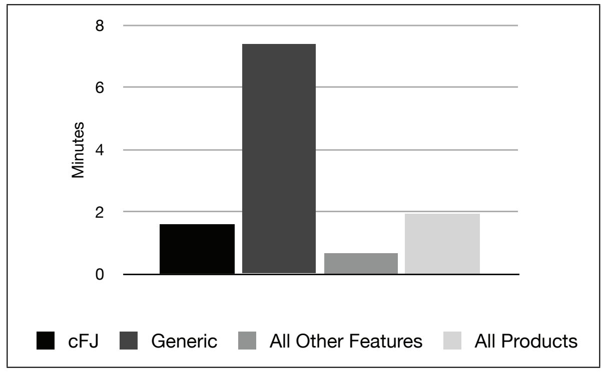

Figure 16: Certification Times for Feature Modules and Language Variants.

6. Adding New Features

A familiar set of issues is encountered when a new feature is added to a product line. Ideally, a new feature would be able to simply update the existing definitions and proofs, allowing language designers to leverage all the hard work expended on formalizing the original language. Some foresight, called *domain analysis* [21], allows language designers to predict VPs in advance, thus enabling a smooth and typically painless addition of new features. What our work shows is a path for the structured evolution of languages. But of course, there is no magic when unanticipated features are added – additional engineering is required.

Existing definitions can be extended and reused *as long as* they already have the appropriate VPs and their inductive definitions left open. For example, once class definitions have a variation point inserted for interfaces, the same VP can also be extended with type parameters for generics. Similarly, once the definition of subtyping has been left open, both interfaces and generics can add new rules for the target language. The important observation is that a definition only has to be enabled for extension precisely once.

Proof reuse is almost as straightforward: as long as an extension is compatible with the set of assumptions in a proof's interface, the proof can be reused directly in the final language. A new feature is responsible for providing the proofs that its extension satisfies the assumptions of the original base language.

Refactoring is necessary when a new feature requires VPs that are not in existing features. A feature which

[3] The source for these features can be found at `http://www.cs.utexas.edu/users/bendy/MMMDevelopment.php`.

makes widespread changes throughout the base language (i.e. Generics), will probably make changes in places that the original feature designer did not anticipate. In this situation, as mentioned in Section 2.1, existing features have to be refactored to allow the new kind of extension by inserting variation points or breaking open the recursion on inductive definitions. Any proofs over the original definition may have to be updated to handle the new extensions, possibly adding new assumptions to its interface.

Feature modules tend to be inoculated from changes in another, unless they reference the definitions of another feature. This only occurs when two features must appear together: modules which resolve feature interactions, for example, only appear when their base features are present. Thus, it is possible to develop an enhanced language incrementally: starting with the base and then iteratively refactoring other features, potentially creating new modules to handle interactions. Once a new feature has been fully integrated into the feature set, all of the previous languages in the product line should still be derivable. If two features F and G commute (i.e. $\mathtt{F} \cdot \mathtt{G} = \mathtt{G} \cdot \mathtt{F}$) their integration comes for free as their interaction module is empty (i.e. $\mathtt{F} \# \mathtt{G} = \mathtt{1}$).

A new feature can also invalidate the assumptions of an existing feature's proofs. In this case, assumptions might have to be weakened and the proof refactored to permit the new extension. Alternatively, if an extension breaks the assumption of an existing proof, the offending feature can simply build a new proof of that lemma. This proof can then be utilized in any other proofs which used that lemma as an assumption, reusing the original proof with the new lemma swapped in. In this manner, each proof is inoculated against features which break the assumptions of other lemmas.

But again, all of this is just another variation of the kinds of problems that are encountered when one generalizes and refactors typical object-oriented code bases.

7. Related Work

The Tinkertype project [16] is a framework for modularly specifying formal languages. Features consist of a set of variants of inference rules with a feature model determining which rule is used in the final language. An implementation of these ideas was used to format the language variants used in Pierce's Types and Programming Languages [23]. This corresponds to our notion of case introduction. Importantly, our approach uses variations points to allow variations on a single definition. This allows us to construct of a single generic proof which can be specialized for each variant, as opposed to maintaining a separate proof for each variation. Levin et al. consider using their tool to compose handwritten proofs, but these proofs must be rechecked after composition. In contrast, we have crafted a systematic approach to proof extension that permits the creation of machine-checkable proofs. After a module's proofs are certified, they can be reused without needing to be rechecked. As long as the module's assumptions hold, the proofs are guaranteed to hold for the final language.

Stärk et. al [26] 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 complier, 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 [1], noting that the proof could also have been developed incrementally. While we present the incremental development of the formal specification of a language here, many of the ideas are the same. An important difference is that our work focuses on structuring languages and proofs for mechanized proof assistants, while the development proposed by [1] is completely by hand.

The modular development of reduction rules are the focus of Mosses' Modular Structural Operational Semantics [19]. 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:

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

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

$$\frac{\mathtt{e} \xrightarrow{\{\mathtt{p}=\mathtt{p}_1[\mathtt{p}_0]\ldots\}} \mathtt{e}'}{\mathtt{let\ p_0\ in\ e} \xrightarrow{\{\mathtt{p}=\mathtt{p}_1\ldots\}} \mathtt{let\ p_0\ in\ e}} \qquad \text{(R-LetE)}$$

These abstract labels correspond to the abstract contexts used by the cFJ subtyping rules to accommodate the Generic features updates. In the same way that R-LetE depends on the existence of a store in the final language, S-Var requires the final context to support a type lookup operation. Similarly, both R-LetB and S-Trans pass along the abstract labels / contexts from their subjudgements.

Both Boite [8] and Mulhern [20] consider how to extend existing inductive definitions and reuse related proofs in the Coq proof assistant. Both only consider the introductions and case extension and also 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. Furthermore, as a result of our formulation, it is possible to check the proofs in each feature module independently, with no need to recheck proof terms when composing features.

Chlipala [9] proposes a using adaptive tactics written in Coq's tactic definition language LTac [10] 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 approach, each refinement was incorporated into a new monolithic language, with the new variant having a distinct set of proofs to maintain. Our feature modules avoid this problem, as each target language derives its proofs from a uniform base, with no need to recheck the proofs in existing feature modules when composing them with a new feature. Adaptive proofs could also be used within our feature modules to make existing proofs robust in to the addition of new syntax and semantic extension points.

8. Conclusion

Mechanically verifying artifacts using theorem provers can be hard work. The difficulty is compounded when verifying all the members of a product line. Features, transformations which add a new piece of functionality, are a natural way of decomposing the building blocks of a product line. Decomposing proofs along feature boundaries enables reuse of proofs from a common base for each target product. These ideas have a natural expression in the evolution of formal specification of programming languages, using the syntax, semantics, and metatheoretic proofs of a language as the core representations. In this paper, we have shown how introductions and Variation Points can be used to structure product lines of formal language specifications.

As a proof of concept, we have used this approach to implement features modules that enhance a variant of Featherweight Java in the Coq proof assistant. Our implementation uses the standard facilities of Coq to build the composed languages. Coq is able to mechanically check the proofs of progress and preservation for the composed languages, which reuse pieces of proofs defined in the composed features. Each extension allows for the structured evolution of a language from a simple core to a fully-featured language. Harnessing these ideas in a mechanized framework transforms the mechanized formalization of a language from a rigorous check of correctness into an important vehicle for reuse of definitions and proofs across a family of related languages.

Acknowledgments. This work was supported by NSF's Science of Design Project CCF 0724979.

References

[1] D. Batory and E. Börger. Modularizing theorems for software product lines: The jbook case study. *Journal of Universal Computer Science*, 14(12):2059–2082, 2008.

[2] D. Batory, Rich Cardone, and Y. Smaragdakis. Object-oriented frameworks and product-lines. In *SPLC*, 2000.

[3] D. Batory, J. Kim, and P. Höefner. Feature interactions, products, and composition. Submiited 2011.

[4] D. Batory, J.N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE TSE*, 30, June 2004.

[5] Don Batory. Feature models, grammars, and propositional formulas. *Software Product Lines*, pages 7–20, 2005.

[6] Don Batory, Rich Cardone, and Yannis Smaragdakis. Object-oriented framework and product lines. In *SPLC 2000*, pages 227–247.

[7] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Springer-Verlag, Berlin, 2004.

[8] Olivier Boite. Proof reuse with extended inductive types. In *Theorem Proving in Higher Order Logics*, pages 50–65, 2004.

[9] Adam Chlipala. A verified compiler for an impure functional language. In *POPL 2010*, January 2010.

[10] 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.

[11] Feature oriented programming. `http://en.wikipedia.org/wiki/Feature_Oriented_Programming`, 2008.

[12] Georges Gonthier. In Deepak Kapur, editor, *Computer Mathematics*, chapter The Four Colour Theorem: Engineering of a Formal Proof, pages 333–333. Springer-Verlag, Berlin, Heidelberg, 2008.

[13] 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.

[14] K.C. Kang. Private Correspondence, 2005.

[15] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52:107–115, July 2009.

[16] Michael Y. Levin and Benjamin C. Pierce. Tinkertype: A language for playing with formal systems. *Journal of Functional Programming*, 13(2), March 2003. A preliminary version appeared as an invited paper at the *Logical Frameworks and Metalanguages Workshop (LFM)*, June 2000.

[17] M. D. McIlroy. Mass-produced software components. *Proc. NATO Conf. on Software Engineering, Garmisch, Germany*, 1968.

[18] R. Monson-Haefel. *Enterprise Java Beans*. O'Reilly, 3rd edition, 2001.

[19] Peter D. Mosses. Modular structural operational semantics. *J. Log. Algebr. Program.*, 60-61:195–228, 2004.

[20] Anne Mulhern. Proof weaving. In *Proceedings of the First Informal ACM SIGPLAN Workshop on Mechanizing Metatheory*, September 2006.

[21] J. Neighbors. The draco approach to constructing software from reusable components. *IEEE TSE*, September 1984.

[22] D.L. Parnas. On the design and development of program families. *IEEE TSE*, SE-2(1):1 – 9, March 1976.

[23] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[24] Y. Smaragdakis and D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM TOSEM*, December 2001.

[25] 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.

[26] Robert Stärk, Joachim Schmid, and Egon Börger. Java and the java virtual machine - definition, verification, validation, 2001.

[27] Rok Strnisa, Peter Sewell, and Matthew J. Parkinson. The Java module system: core design and semantic definition. In *OOPSLA*, pages 499–514, 2007.

[28] Michael VanHilst and David Notkin. Decoupling change from design. *SIGSOFT Softw. Eng. Notes*, 21:58–69, October 1996.

A. Open Inductive Definitions in Coq

Figure 17 shows a concrete example of crafting an extensible inductive definition in Coq. The target language of `FJ` = `Cast` · `cFJ` is built by importing the Coq modules for features `Cast` and `cFJ`. The target syntax is defined as a new data type, `E`, with data constructors `cFJ` and `Cast` from each feature. Each constructor wraps the syntax definitions from their corresponding features, closing the inductive loop by instantiating the abstract parameter `E'` with `E` , the data type for the syntax of target language.

These parameters also affect data types which reference open inductive definitions. In particular, the signature of typing rules and the transition relation are now over the parameter used for the final language. Of course, these rules are defined over the actual syntax definitions given in a feature module. In order for the signatures to sync up, these rules are parameterized over a function that injects the syntax defined in the feature module into the syntax of the final language. Since the syntax of a module is always included alongside its typing and reduction rules in the target language, such an injection always exists.

```
Inductive E (E' : Set): Set :=
 | e_var : Var → E
 | fd_access : E' → F → E
 | m_call : E' → M → List E' → E
 | new : Ty → List E' → E.
```

cFJ.v

```
Inductive E (E' : Set) : Set :=
 | e_cast : Ty → E' → E.
```

cast.v

```
Require Import cFJ.
Require Import cast.
Inductive E : Set :=
 | cFJ : cFJ.E E → E
 | cast : cast.E E → E.
```

FJ.v

Figure 17: Syntax for cFJ and Cast Features and Their Union.

Parameterization also allows feature modules to include VPs as shown in Figure 18. The VPs in each module are explicitly represented as abstract sets/predicates/functions, as with the parameter `TP_m` used to extend the expression for method calls in `cFJ.v`. Other features can provide appropriate instantiations for this parameters. In Figure 18, for example, `FGJ.v` builds the syntax for the target language by instantiating this VP with the definition of `Generic_VP` given in `Generic.v`.

```
Definition TP_m := unit.
Inductive cFJ_E (E : Set) (TP_m : Set): Set :=
 | e_var : Var → cFJ_E
 | fd_access : E → F → cFJ_E
 | m_call : TP_m → E → M → List E → cFJ_E
 | new : C → List E → cFJ_E.
```

cFJ.v

```
Definition TP_m := List Ty.
```

Generic.v

```
Require Import cFJ.
Require Import Generic.
Definition TP_m := Generic.TP_m.
Inductive E : Set :=
 | cFJ : cFJ_E E TP_m → E
```

FGJ.v

Figure 18: Coq Syntax for cFJ with a Variation Point, and its instantiation in FGJ.

A.1 Feature Composition in Coq

Each feature module is implemented as a Coq file which contains the inductive definitions, variation points, and proofs provided by that feature. These modules are certified independently by Coq. Once the feature modules have been verified, a target language is built as a new Coq file. This file imports the files for each of the features included in the language, e.g. "`Require Import cFJ.`" in Figure 17. First, each target language definition is built as a new inductive type using appropriately instantiated definitions from the included feature modules, as shown in Figures 17 and 18. Proofs for the target language are then built using the proofs from the constiuent feature modules per the discussion in section 4.6, as shown in Figure 13. Proof composition requires a straightfoward check by Coq that the assumptions of each feature module are satisfied, i.e. that a feature's interface is met by the target language. Currently each piece of the final language is composed by hand in this straightforward manner; future work includes automating feature composition directly.