Feature Modularity in Mechanized Reasoning

*Ph.D. Proposal*

Benjamin Delaware

Department of Computer Science
University of Texas at Austin
bendy@cs.utexas.edu

**Abstract.** One common and effective approach to reuse is to decompose a system into modules representing *features*. New variants can then be built by combining these features in different ways. This thesis proposes that proofs establishing semantic properties of a system can be similarly decomposed and reused to prove properties for novel feature combinations. Features can cut across the standard modularity boundaries, presenting a fundamental challenge to modular reasoning. The proposed contributions are threefold:

1. Showing how the mechanized syntax, semantics and meta-theory proofs of a programming language can be effectively modularized into features that can be composed in different ways to build programming languages with fully mechanized meta-theory.
2. Demonstrating how modularization of semantic properties alongside definitions enables efficient reasoning about an entire family of programs built from a common set of features.
3. Investigating how that these techniques can aid in the semantically correct composition of interpreters for different languages.

**1 Introduction**

Mechanized theorem proving can be quite hard: large-scale proof developments take multiple person-years and consist of tens of thousand lines of proof scripts [23, 27]. Given the effort invested in formal verification, it is desirable to reuse as much of an existing formalization as possible when modifying it or extending it with new features. One common and effective approach to reuse is to decompose a system into modules which can then be combined with new features in novel ways. Allowing researchers to share language designs and facilitating experimentation were two key reasons the authors of the PoplMARK challenge [21] identified *reuse of components* as a key challenge in mechanizing programming language meta-theory.

Modularization of programming language proofs is a challenging problem precisely because the natural decomposition of a language cuts across standard modularity boundaries. Reuse is commonly achieved by copying an existing formalization and manually adapting it to incorporate new features, evoking (and suffering from the same drawbacks as) the cut-paste-patch approach to...
code reuse in software engineering. An extreme case of this cut-paste-patch approach can be found in Leroy’s three person-year CompCert verified compiler project [27]. CompCert consists of eight intermediate languages in addition to the source and target languages, many of which are minor variations of each other.

The programming language literature is replete with these sorts of language variations. Taking a core language such as Featherweight Java (FJ) [25] 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 similar cut-paste-patch approach to proof reuse. Patching up proofs composed in this manner requires care, making integration of new features something of an art; this is one reason why small core languages are typically extended. One of the benefits of modularization is to make composition less ad-hoc, potentially allowing for cleaner integration of new features. Modularization transforms mechanized semantics from an important confidence-booster into a powerful vehicle for reuse.

Component-based reuse has long been studied in the software engineering community, with cross-cutting features being particularly useful in the implementation of the families of related programs known as Software Product Lines (SPLs). The potentially exponential number of products in a SPL family makes efficiency key for product-line analyses which reason about each member. Reducing a proof about a product to proofs about its constituent features is an effective way to efficiently scale product line analyses. Finding this decomposition and developing techniques for reasoning about cross-cutting features are both important challenges in SPL engineering.

This work sits squarely at the intersection of programming languages, software engineering, and theorem proving and proposes contributions in all three areas. This thesis proposes that the mechanized syntax, semantics and meta-theory proofs of a programming language can be effectively modularized into features which can be composed to build programming languages with fully mechanized meta-theory. In addition, it proposes that modularization of semantic properties enables efficient reasoning about the entire family of products that can be built from these definitions. Finally, it proposes to leverage these techniques to aid in the semantically correct composition of interpreters.

2 Feature Modularization of Mechanized Meta-Theory

Modularity is not a foreign concept in theorem proving– the eleven phases of Leroy’s certified compiler are each independently verified components. The issue with modularizing the mechanized formalization of a programming language into features is that features can cut across the standard modularity boundaries of theorem provers. Consider the (pen-and-paper) definitions of the FJ calculus [25] presented in the left-hand column of Figure 1. These definitions are a subset of
the *syntax* in which FJ programs are written and the *semantics* that governs their behavior.

<table>
<thead>
<tr>
<th>FJ Expression Syntax</th>
<th>FGJ Expression Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>e ::= x</td>
<td>e ::= x</td>
</tr>
<tr>
<td>e.f</td>
<td>e.f</td>
</tr>
<tr>
<td>e.m (e)</td>
<td>e.m (T) (e)</td>
</tr>
<tr>
<td>new C(e)</td>
<td>new C(T) (e)</td>
</tr>
<tr>
<td>(C) e</td>
<td>(C (T) e)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FJ Subtyping</th>
<th>T &lt;= T</th>
</tr>
</thead>
</table>
| S <= T  
S <= V | (S-TRANS) |
| T <= T | (S-REFL) |

<table>
<thead>
<tr>
<th>FGJ Subtyping</th>
<th>Δ^δ ⊢ T &lt;= T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Δ ⊢ X &lt;= Δ(X) α</td>
<td>(GS-VAR)</td>
</tr>
</tbody>
</table>
| Δ^δ ⊢ S <= T  
Δ^δ ⊢ T <= V | (GS-TRANS) |
| Δ^δ ⊢ S <= V | (GS-REFL) |

<table>
<thead>
<tr>
<th>class C extends D {...}</th>
<th>C &lt;= D</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>class C (X &lt;= T) extends D V {...}</th>
<th>C &lt;= T</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>FJ New Typing</th>
<th>Γ ⊢ e : T</th>
</tr>
</thead>
</table>
| fields(C) = V T  
Γ ⊢ e : U  
U <= V | Γ ⊢ new C(U) : C |
| (T-NEW) | (GT-NEW) |

<table>
<thead>
<tr>
<th>FGJ New Typing</th>
<th>Δ^δ Γ ⊢ e : T</th>
</tr>
</thead>
</table>
| Δ ⊢ C(T) γ  
fields(C (T) (T) = V T | Δ^δ Γ ⊢ e : U  
Δ^δ ⊢ U <= V |

| Δ^δ Γ ⊢ new C(T) (e) : C | (GT-NEW) |

Fig. 1: Selected FJ Definitions with FGJ Changes Highlighted

The right hand column highlights how these definitions can be extended to build the FGJ calculus. The changes are not completely ad-hoc and can be broadly categorized:

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

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

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.

The addition of generics makes changes that are woven throughout the existing syntax and semantics of FJ. The standard approach to formalizing programming languages uses closed definitions which do not support extension. The proofs forming the meta-theory of a language ensuring the semantics are consistent (e.g. type soundness) are written over these closed definitions. The lack of extensibility forces a cut-paste-patch approach to reuse of both definitions and proofs. The first contribution of this thesis is a framework for defining and composing language features, enabling a more structured approach to reuse of mechanized proofs.

2.1 Feature Modularization and Composition

Conceptually, the two languages in Figure 1 are compositions of the FJ and Generic features, succinctly written as expressions of an abstract algebra with binary composition operator $+: FJ = FJ$ and $FGJ = FJ + Generic$. As the set of language features grows (e.g. with the addition of an Interface feature Interface), so does the number and complexity of possible languages. Given an operation $\delta$ which maps algebraic expressions to definitions, the two expressions build the languages given in Figure 1. One (inefficient) strategy for constructing $\delta$ is as a 1-1 mapping by simply formalizing each possible language individually, which of course greatly limits reuse.

Full modularization of a language amounts to making $\delta$ distribute: $\delta(FJ + Generic) = \delta(FJ) +_\delta \delta(Generic)$. Modularizing the syntax and semantics of a language depends on both developing reusable language components ($\delta(\cdot)$) and composition operators ($+_\delta$) that can assemble complete definitions from components. There have been a number of previous approaches to modularizing syntax and semantics that embed the components in a variety of programming languages: Semantic Lego [20] used Lisp, while Datatypes a lá Carte [43] used Haskell.

The focus of this dissertation is another operation, $\pi$, for modularization and composition of proofs. This operator builds the meta-theory of a language. None of the previous approaches consider proof composition. (The Tinkertype [28] project does consider composition of handwritten proofs, but without the associated semantics, this is effectively $\delta$-composition). Modularizing fully mechanized language formalizations into features depends on both operators: $\delta$ is used for syntax and semantics, while $\pi$ builds the meta-theory.
This first contribution of this thesis is to show how to realize π operation in the Coq proof assistant [7] through the decomposition of proofs into reusable proof components [17] and an proof composition operator [18]. We have implemented our approach as framework for building and composing modular definitions and proofs in Coq called Meta-Theory à la Carte (MTC).

2.2 Modular Definitions

To reason about the syntax and semantics components, δ also has to be defined within the proof assistant. The lack of general recursion in proof assistants prevents a direct port of existing approaches to the modularization of syntax and semantics.

Implementing the final four changes of Figure 1 as instantiation of variation points (VPs) in existing definitions reduces the five categories of changes into two simple operations. Variation points are a standard concept in product line designs [3]. In this setting, they are parameters representing where variation can occur and which must be instantiated to build a concrete language.

As an example, Figure 2a shows the the FJ expression grammars with the VPs TP\textsubscript{m} and TP\textsubscript{t} added. Inlining the instantiations of Figure 2b produces the grammar of FJ while Figure 2c builds the grammar of FGJ. Figure 3 demonstrates how judgements can be similarly extended to produce typing rules for both FJ and FGJ.

```
\textbf{Fig. 2: Modification of grammars through VP instantiation.}
```

```
e ::= x
   | e.f
   | e.m \texttt{TP}_m (\bar{e})
   | \texttt{new C TP}_t (\bar{e})
   | (\texttt{C TP}_t) e ;

\begin{align*}
\texttt{TP}_n &::= c; \\
\texttt{TP}_t &::= c;
\end{align*}
\begin{align*}
\texttt{TP}_n &::= (\bar{T}); \\
\texttt{TP}_t &::= (\bar{T});
\end{align*}
```

Typically syntax and semantics are embedded in Coq as inductive datatypes. A program is a datatype member, as are typing and reduction derivations. Accordingly, we need operations to extend inductive datatypes with new cases and to instantiate VPs in order to support the extensions of Figure 1. The latter operation is supported quite naturally through Coq’s abstraction mechanisms: a VP is simply a parameter of a definition which is instantiated appropriately. Case addition requires more care, however.

The fundamental issue is that case addition requires modifying inductive structures, but standard inductive datatypes are closed. Wrapping inductive datatypes doesn’t work: the combined BNF rule \( E ::= E_1 \mid E_2 \) doesn’t allow \( E_2 \)
expressions to be used in $E_1$ expressions or vice versa. A category theoretic presentation of datatypes provides an elegant foundation for solving this problem.

Functors provide a unifying framework for describing datatypes. A functor $F$ is simply an operation mapping objects to objects and arrows (functions) to arrows which respects composition:

$$F(f \circ g) = F(f) \circ F(g)$$

and identity:

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

The datatype $\mu_F$ “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 separates datatype description (using a functor) from datatype definition (taking the least fixed point). The two are combined in Coq’s datatype definition mechanism. Importantly, functors can be modified and composed to build different datatype “descriptions”. Figure 4 presents an example of this formulation. The $\text{Arith}$ and $\text{Bool}$ functors define simple Arithmetic and Boolean expressions. We can then take the fixpoint of either function with $\text{in}$ to build the syntax of arithmetic ($\text{Exp}_1$) or boolean ($\text{Exp}_2$) expression languages. $\text{FPlus}$ is a generic operation for combining two functors which $\text{Exp}_3$ uses to build the syntax of a combined boolean and arithmetic expression language. Similar techniques can modularize semantic judgements using indexed functors.

Missing from Figure 4 is a definition of $\text{in}$, and for good reason, as it cannot be defined within Coq! A least fixpoint operator for datatypes could be used to build inductive datatypes that violate the positivity condition, jeopardizing the soundness of the theorem prover. Coq’s standard inductive datatype definition mechanism enforces this using a syntactic check that an occurrence of an inductive type never appears in a negative position. Without this check, the untyped lambda calculus could be embedded in Coq, from which $\bot$ could be derived. A general $\text{in}$ could be applied to a definition with a negative occurrence of its parameter, breaking the soundness of Coq.
\begin{verbatim}
Inductive Arith (A : Set) :=
Lit : nat → Arith A | Add : A → A → Arith A.
Inductive Bool (A : Set) :=
BLit : bool → Bool A | If : A → A → A → Bool A.

Definition FPlus (F G : Set → Set) : Set → Set :=
fun A : Set ⇒ {F A} + {G A}.
Definition Exp1 := in Arith.
Definition Exp2 := in Bool.
Definition Exp3 := in (FPlus Arith Bool).
\end{verbatim}

Fig. 4: Product line of simple expression languages.

One direct solution is to use the standard inductive datatype definition mechanism to define a wrapper type for each functor, closing the induction by hand. Because the functors are known, the positivity condition can be syntactically checked. This is effectively defining a ad-hoc in operator \cite{17} for each functor, which quickly leads to a large amount of boilerplate code. It also prevents generic definitions of functions. MTC implements an alternate approach that uses church-encoded datatypes\cite{18} which do not suffer from the positivity restrictions.

2.3 Modular Proofs

Functions from the fixpoint of a functor \( F \) to a type \( A \) can be built as folds of \( F \)-algebras, or morphisms from \( F(A) \) to \( A \): \( f :: F(A) → A \). Since \( µ_F \) is the initial object in the category of \( F \)-algebras, for each \( F \)-algebra \( f \) there exists a unique function, \( \text{fold}_f : µ_F → A \), such that:

\[
\text{fold}_f \circ \text{in} = f \circ F(\text{fold}_f)
\]

Expressed as a commuting diagram:

\[
\begin{array}{c}
F(µ_F) \xrightarrow{\text{in}} µ_F \\
F(\text{fold } f) \downarrow \quad \downarrow \text{fold } f \\
F(A) \xrightarrow{f} A
\end{array}
\]

Figure 5 presents evaluation algebras that build interpreters for the languages in Figure 4. Importantly, there is a generic operator \texttt{FAlgebraPlus} for building composite \texttt{“FPlus”}-algebras from components. In the same way that \texttt{FPlus} is the composition operator for functors, \texttt{FAlgebraPlus} is the composition operator for \( F \)-algebras. The definition of \texttt{fold} depends on the method used to build the fixpoint of the corresponding functor. If the per-definition approach is used,
fold is also defined on a per-algebra basis as the fixpoint which dispatches to the constituent algebras. Recursive application is embedded in Church-encoded datatypes, so fold is simply application.

```
Definition evalArith (fa : Arith val) :=
  match fa with
  | Lit n ⇒ n | Add m n ⇒ m + n
  end.

Definition evalArith (fa : Bool val) :=
  match fa with
  | BLit b ⇒ b | If c t e ⇒ if c then t else e
  end.

Definition FAlgebraPlus (F G : Set → Set) (FAlgebra : F val → val) (GAlgebra G val → val) :
  (FPlus F G) val → val :=
  fun A : Set ⇒ (F A) + {G A}.

Definition eval1 := fold evalArith.
Definition eval2 := fold evalBool.
Definition eval3 := fold (FAlgebraPlus evalArith evalBool).
```

Fig. 5: Defining interpreters for the languages of Figure 4.

Since proofs in Coq are essentially functions with particularly rich datatypes, they can also be expressed as algebras with an analogous composition operator \( + \). The choice of datatype encoding has an important impact on the signature of these algebras, however. The standard encoding presents no trouble. Defining induction principles for Church-encoded datatypes without resorting to axioms, on the other hand, has long been an open problem and was the reason inductive datatypes were introduced to Coq[37]. We have shown that such principles can be recovered through a novel axiom-free approach based on an adaptation of the Universal Property[24] of folds. This allows proof algebras to be used to reason about church encoded datatypes and is the foundation for modular reasoning in MTC.

2.4 Proof Interfaces

Actually building reusable proof components adds an important wrinkle to the mix: the notion of proof interfaces[17]. In order to build reusable components, each feature must abstract over the functors and VPs defining the final language. Case analysis and inversion principles, which are absolutely standard proof techniques, are not available when reasoning about these parameters. In order to reason about these abstractions, a proof algebra must constrain the set
of possible functors and VPs through a set of assumptions about their properties. These assumptions move the case analysis to the abstract super-functor and form an interface for each proof algebra. This is effectively an object-oriented style dispatch.

Building meta-theory proofs for a language now depends on ensuring that the interface of each proof algebra component is satisfied by checking for proofs of each assumption. The algebras used to build these proofs may themselves have assumptions which need to be discharged. MTC uses typeclasses and Coq’s auto tactic to automate this interface check as a backtracking search using a database of proof algebras.

These interfaces naturally expose feature interactions[6]—proofs and definitions that are only required when two features are composed together. Feature interaction detection and resolution are important challenges in SPLs. A more precise definition of feature interactions covers both domains: given a specification of the composition of two features, $\text{Spec}(F + G)$, a feature interaction is the (possibly empty) feature $F\#G$ that makes the composition satisfy its specification:

$$F + G + F\#G \models \text{Spec}(F + G)$$

Feature interactions can be difficult to detect in SPLs, due to both lack of specification and appropriate analyses. In the theorem proving setting, the task is much simpler. Since a language’s meta-theory makes up a complete semantic specification, missing feature interactions are simply missing proof pieces. These are automatically detected during a proof search. If there is no proof of an assumption in the fact database, some interactions can be automatically resolved through the use of custom tactics. Thanks to proof irrelevance, the efficiency of the generated proof is unimportant.

3 SPL Reasoning with Features

Defining reusable proof components is particularly useful when checking whether a property holds for every member of a feature-based product line. Doing so has two important benefits:

1. Features can be reused in multiple products, avoiding repeated work. For example, $\pi(F)$ can be developed once, and then used to establish both $\pi(F + G)$ and $\pi(F + H + K)$.
2. Multiple products can take advantage of common feature selections to efficiently check satisfaction of a module’s proof interface for each product.

These benefits can be exploited to efficiently check properties of SPLs.

3.1 Safe Composition of Software

SPLs usually have a feature model(FM) describing the desired set of feature selections that make up the product line. One standard SPL analysis is ensuring
that all feature selections permitted by the feature model compose into well-formed programs, also known as checking safe composition:

$$\forall G \subseteq F, \text{FM}(G) \rightarrow \vdash \text{Compose}(G) \text{ OK}$$

(3)

Of course, it is possible to type-check individual programs by building and then compiling them. A product line can have thousands of programs, making enumeration a highly inefficient strategy for ensuring that all legal programs are type-safe (this is true of any product line analysis). Efficiently checking safe composition requires a novel approach to type checking.

Our approach relies on separating type-checking into a constraint generation phase and a constraint satisfaction phase. This separation allows a feature to be type-checked in isolation, delaying satisfaction of its constraints until the feature is included in a complete program. The soundness of this approach relies on two important lemmas:

**Lemma 31 (Safety of Constraint-Based Typing)** If a program is well-formed subject to a set of constraints, and that program satisfies those constraints, that program is well-formed under the standard typing rules.

$$\vdash P : C \rightarrow P \vdash C \rightarrow \vdash P \text{ OK}.$$  

Lemma 31 establishes that the two-phased, constraint-based type system is sound with respect to the standard type system.

**Lemma 32 (Monotonicity of Features Constraints)** If a set of features $F_i$ is well-formed subject to a set of constraints $C_i$, its composition is a program that is typeable under the constraints $C$, and those constraints are a subset of $C_i$.

$$\forall i, \vdash F_i : C_i \rightarrow \exists C, \vdash \text{Compose}(F_i) : C \land C \subseteq \bigcup_i C_i.$$  

Lemma 32 shows that the constraints generated for a program built from a given feature selection are a subset of those generated for each included feature. Taken together, the two proofs show that satisfaction of the constraints of a product’s constituent features ensures that a program is well-formed. Type safety of the approach follows immediately:

**Theorem 33 (Type Safety for Feature Typing)** If a base set of features $F_i$ is well-formed subject to a set of constraints $C_i$, and if the composition of every valid selection of features satisfies the constraints of its constituent features, then every valid feature selection builds a well-typed program.

$$\forall i, \vdash F_i : C_i \rightarrow$$

$$\forall G \subseteq F, \text{FM}(G) \rightarrow \text{Compose}(G) \models \bigcup_i C_i \rightarrow$$

$$\forall G' \subseteq F, \text{FM}(G') \rightarrow \vdash \text{Compose}(G') \text{ OK}.$$
Checking $\text{Compose}(G) \equiv \bigcup_i C_i$ is equivalent to checking the validity of a propositional formula. Given a constraint, each feature in the product line is analyzed to see if including it in a product would satisfy that constraint. A SAT clause is built 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, and the SAT solver continues searching under the newly constrained feature selection. In this manner, a SAT solver can easily check whether a valid feature selection satisfies all of the clauses generated by the included features. Checking validity of this formula is equivalent to checking whether its negation is satisfiable. Thus, (3) has been reduced to checking $\exists G, \text{FM}(G) \land \neg\text{Propify}(G)$. This problem is decidable, and SAT-solvers are quite good at efficiently solving such formulas.

Table 1 presents the runtimes of a tool based on this approach. The tool 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, less than the time it took to generate and compile a single program in the product line.

<table>
<thead>
<tr>
<th>Product Line</th>
<th># of Features</th>
<th># of Code Base</th>
<th>Program Line</th>
<th>Jak/Java LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPL</td>
<td>7</td>
<td>20</td>
<td>2000/2000</td>
<td>1K/1K</td>
</tr>
<tr>
<td>BPL</td>
<td>17</td>
<td>8</td>
<td>12K/16K</td>
<td>8K/12K</td>
</tr>
<tr>
<td>GPL</td>
<td>18</td>
<td>80</td>
<td>1800/1800</td>
<td>700/700</td>
</tr>
<tr>
<td>JPL</td>
<td>70</td>
<td>56</td>
<td>34K/48K</td>
<td>22K/35K</td>
</tr>
</tbody>
</table>

Table 1: Product Line Statistics from [44].

Just as with proof algebra components, well-formedness of a software feature module cannot be completely established in isolation. Type-checking depends on querying the product that a feature is included in. In order to separate constraint generation from satisfaction, an interface is needed to abstract the final product way. This means that composition of typing derivations can fail when the interface is not satisfied. At its core, the goal of safe composition is to ensure that the semantic interface of each feature is satisfied by the composition of every feature selection allowed by the feature model.

### 3.2 Semantic Interfaces

The proof interfaces of Theorem Product Lines and the typing constraints of Software Product Lines provide this interface. In both cases, inclusion of a feature places constraints on the final product, allowing the verification of a feature that is independent of a specific product in which it is included. The typing constraints
of SPLs specify the signatures of methods, fields and classes that must be present in the product in order for the structure being typed to be well-formed. These constraints are also a kind of interface which specifies the shape of programs in which the structure can be safely included.

In both cases, a proof guarantees that satisfaction of the constraints entails a desired property, i.e. they form a semantic interface. Lemma 31 ensures that checking that a single program is well-typed reduces to checking that the program satisfies the constraints of its constituent features. The analog for TPLs is a proof of consistency for Coq’s underlying logic. This is actually a very powerful property: \texttt{FAlgebraPlus} can compose proofs of arbitrary properties, so there is no need for a proof of correctness for each individual property. Soundness for any proof built from \texttt{FAlgebraPlus} is immediate by virtue the search and proof construction for TPLs being done entirely within the theorem prover.

\section{Current Contributions}

The work on proof modularization outlined in Section 3 has produced two papers\cite{17}\cite{18}, the first of which appeared in OOPSLA '11 and the second of which is to be published in POPL'13. Both papers are attached as appendices, but to summarize their technical contributions:

- An approach to language modularization based on case extension and variation point instantiation, formalized as functor and proof algebra composition.
- A formulation of language modules as functors and algebras using church-encoded datatypes to avoid positivity problems.
- A novel axiom-free approach to reasoning about church-encoded datatypes using proof algebras and the Universal Property of folds.
- The MTC framework for the Coq proof assistant, which uses church-encodings for extensibility.
- The development of mediating typeclass instances for integrating existing algebras into languages with higher-order features.
- Two case studies of modular mechanized metatheory in Coq using two different approaches to formalizing semantics:
  - \texttt{GifJ} Featherweight Generic Java with interfaces using standard typing judgements and reduction rules for its semantics and featuring an intricate proof of soundness.
  - \texttt{miniML} A ML-like core language with binders and general recursion using interpreters for a denotational-style semantics.

The work on safe composition of product lines of Section 3 was published \cite{16} in FSE '09 and is also attached as an appendix. To summarize its technical contributions:

- A formalization of a core language for feature composition called Lightweight Feature Java.
- A proof of safe composition for the constraint-based type system of LFJ.
- Full mechanizations of the LFJ calculus and the proof of safe composition in the Coq type assistant.
5 Proposed Work

The proposed work is to implement an efficient check of safe composition of product lines of theorems and to study semantically correct interpreter composition using MTC.

5.1 Unifying Safe Composition

At its core, the goal of safe composition is to ensure that the semantic interface of each feature is satisfied by the composition of every feature selection it is allowed to be in. I propose to investigate safe composition for product lines of theorems. Preliminary work developing a conditional composition operator has been promising. A thoughtful design of the corresponding proof operator avoids an exponential blowup during proof search. Observations from this investigation should provide insights into product-line analyses for SPLs as well.

Remaining work includes:

– Design and implementation of the conditional operators for VP instantiation.
– Further experimentation by checking safe composition for the two product lines of languages already developed. Checking safe composition of the GiFJ product line in particular might require further optimizations to the search algorithm and should be a good case study of how to scale the approach.

5.2 Interpreter Composition

One approach to model-driven development is through interpretation of models[11]. Integrating interpreters is a challenging problem because they can interact in surprising (and potentially hard-to-detect ways. Our current miniML case study uses interpreters to define semantic functions, suggesting that MTC might provide a good framework for studying interpreter composition.

One proposed contribution is to study the integration of monads into the MTC framework. Polymorphic monads allow for the easy integration of components that use different effects, but reasoning about them is very challenging in the open-functor world of MTC. Solving this is a major contribution that increases the applicability of the MTC framework.

Furthermore, assuming that an interpreter’s correctness is fully specified as a theorem, feature interactions are missing proof pieces which can be caught automatically during proof search. In some cases, they can even be automatically discharged through smart tactics hooked into the search algorithm. Existing work has shown how automatic feature detection can be useful in the realm of SPLs[41], suggesting similar benefits can be achieved for TPLs. I propose to further investigate the application of MTC towards interpreter composition, focusing on automatic detection and resolution of feature interactions.
5.3 Long-term Research: Advanced Composition Operators

One disadvantage of VP instantiation is that inserting appropriate VPs requires domain analysis and foresight. Designing appropriate proof interfaces for VPs involves tradeoffs between extensibility and ease-of-reasoning. While engineering will always be an important part of the reusable component design, additional operators have the potential to decrease the amount of preplanning needed by component designers. These operators should replace VPs as the vehicle for the four categories of changes from Figure 1.

In particular, a product operator $\otimes$ would be a useful complement to $\mathsf{FPlus}$ for functor composition. A analogous operator $\otimes_{\mathsf{ex}}$ should be used to compose proof algebras. The signature of this operator will define the necessary proof updates that a component with a modification must provide.

I anticipate (at least) the following challenges:

– The operator must distinguish between the different sums allowed by a functor, appending appropriate information for each branch. Experimentation has shown that a dependently typed function can do this quite naturally.
– Since these components must name the modified product integrating multiple $\otimes$ modifications is an open questions—after one application, the functor’s “name” has been changed. One potential solution is to merge multiple applications into a single one.
– MTC’s injection framework breaks down under this operator—it is no longer possible for a component to directly project into a superfunctor. Further investigation is needed into how much overhead is needed to work around this.
– Alterations to types can affect types that depend on them, e.g. typing judgements. Updating these will require a new operator as well.

Given these challenges, it is unclear that $\otimes$ can be implemented within the desired time span of this thesis. I propose to tackle some of these challenges in the thesis, making some progress towards its eventual realization.

6 Related Work

The proposed contributions are to both programming languages and software engineering, and the related work also spans both fields.

Modular Language Components An initial datapoint for the structuring of modular language developments was the development of a complete Java 1.0 compiler through incremental refinement of a set of Abstract State Machines [42]. Starting with a core language of imperative Java expressions which contains a grammar, interpreter, and compiler, the authors added features to incrementally derive an interpreter and compiler for the full Java 1.0 specification. The authors then wrote a monolithic proof of correctness for the full language. Later work cast this approach in the calculus of features [5], suggesting that the proof could also
have been developed incrementally. An important difference is that this proposal focuses on structuring languages and proofs for mechanized proof assistants, while the development proposed by [5] is completely by hand.

The modular development of reduction rules are the focus of Mosses’ Modular Structural Operational Semantics (MSOS) [31]. 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:

\[
\begin{align*}
    d \xrightarrow{X} d' \\
    \text{let } d \text{ in } e \xrightarrow{X} \text{let } d' \text{ in } e
\end{align*}
\]  

(R-LETB)

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

\[
\begin{align*}
    e \xrightarrow{\{p=p_1[p_0]...\}} e' \\
    \text{let } p_0 \text{ in } e \xrightarrow{\{p=p_1...\}} \text{let } p_0 \text{ in } e
\end{align*}
\]  

(R-LET)

These abstract labels correspond to the abstract contexts used by the cFJ subtyping rules to accommodate the updates of the Generic feature. In the same way that R-LET requires 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.

MSOS serves as the foundation for the recently proposed Programming Language Components and Specifications project [40]. The project is focused on the development of modular semantics through the use of “highly reusable” language components (called funcons). The ultimate goal of the project is a large library of these funcons. Semantics of higher-level languages is given through a reduction to a collection of funcons; modular verification is not a focus.

**Reuse of Mechanized Meta-Theory** Several tool-based approaches have been developed for modularizing mechanized meta-theory, although none is based on a proof assistant’s modularity features alone. The Tinkertype project [28] is a framework for modularly specifying formal languages. It was used to format the language variants used in Pierce’s “Types and Programming Languages” [39], and to compose traditional pen-and-paper proofs.

Both Boite [9] and Mulhern [32] consider how to extend existing inductive definitions and reuse related proofs in the Coq proof assistant. Both their techniques rely on external tools that are no longer available and extensions are written with respect to an existing specification. As such, features cannot be checked independently or easily reused with new specifications. In contrast, our approach is fully implemented within Coq and allows for independent development and verification of features.
Chlipala [10] proposes a using adaptive tactics written in Coq’s tactic definition language LTac [15] 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 approach avoid this problem, as each target language derives its proofs from a set of independently checked proof algebras. Adaptive proofs could also be used within our feature modules to make existing proofs robust in to the addition of new syntax and semantic variation points.

Extensibility in MTC An important contribution of MTC is the use of universal properties to provide modular reasoning techniques for encodings of inductive data types that are compatible with theorem provers like Coq. Old versions of Coq, based on the calculus of constructions [12], also use Church encodings to model inductive data types [38]. However, the induction principles for reasoning about those encodings had to be axiomatized and, among other problems, they endangered strong normalization of the calculus. The calculus of inductive constructions [37] has inductive data types built-in and was introduced to avoid the problems with Church encodings. MTC returns to Church encodings to allow extensibility but does not use standard induction principles since they are not extensible. Instead, by using a reasoning framework based on universal properties we get two benefits for the price of one: universal properties allow modular reasoning and they can be proved without axioms in Coq.

Our approach to extensibility combines and extends ideas from existing solutions to the expression problem. The type class infrastructure for (Mendler-style) F-algebras is inspired by DTC [19, 43]. However type-level fixpoints, central to DTC, cannot be used in Coq because they require general recursion. To avoid general recursion, we use least-fixpoints encoded as Church encodings [8, 38]. Church encodings inspired other solutions to the expression problem before (especially in object-oriented languages) [35, 33, 34]. However those solutions do not use F-algebras; instead, they use an isomorphic representation called object algebras [34]. Object algebras are a better fit for languages where records are the main structuring construct (such as OO languages). Our solution differs from previous approaches in the use of Mendler-style F-algebras instead of conventional F-algebras or object algebras. Unlike previous solutions to the expression problem, which focus only on the extensibility aspects of implementations, we also deal with modular reasoning and the extensibility aspects of proofs and logical relations.

Semantics and Interpreters A particularly prominent line on modular interpreters is that of using monads to structure semantics. Moggi [30] pioneered monads to model computation effects and structure denotation semantics. Liang et al. [29] introduced monad transformers to compose multiple monads and build
modular interpreters. Jaskelioff et al. [26] used an approach similar to DTC in combination with monads to provide modular implementation of mathematical operational semantics. The proposed work includes incorporating monads into MTC so that it can be used to model more complex language features. However, unlike previous work, MTC also has to consider modular reasoning. Monads introduce important challenges in terms of modular reasoning. Only very recently some modular proof techniques for reasoning about monads have been introduced [36, 22]. While this is a good step forward, it remains to be seen whether these techniques are sufficient to reason about suitably generalized modular statements like soundness.

The above approaches mainly involve pencil-&-paper proofs. Mechanization of interpreter-based semantics clearly poses its own challenges. Yet, it is highly relevant as it bestows the high degree of confidence in correctness directly on the executable artifact, rather than on an intermediate formulation based on logical relations. Danielsson [14] uses the partiality monad, which fairly similar to our bounded fixpoint, to formalize semantic interpreters in Agda. He argues that this style is more easily understood and more obviously deterministic and computable than logical relations. Danielsson does not consider modularization of definitions and proofs, however.

**Product Line Analyses** Much of the existing work on type checking feature-oriented languages has focused on checking a single product specification, as opposed to checking an entire product line. Apel et al. [2] propose a type system for a model of feature-oriented programming based on Featherweight Java [39] and prove soundness for it and some further extensions of the model. gDEEP [1] is a language-independent calculus designed to capture the core ideas of feature refinement. The type system for gDEEP transfers information across feature boundaries and is combined with the type system for an underlying language to type feature compositions.

Thüm et. al [45] 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.

The strategy of representing feature models as propositional formulas in order to verify their consistency was first proposed in [4]. The authors checked the feature models against a set of user-provided feature dependences of the form \( F \rightarrow A \lor B \) for features \( F \), \( A \), and \( B \). This approach was adopted by Czarnecki
and Pietroszek [13] 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 [4], 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.

The previous two approaches relied on user-provided constraints when validating feature models. The genesis of our constraint-based type system was a system developed by Thaker et al. [44] 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. This system relied on a set of rules for generating these constraints with no formal proof showing they were necessary and sufficient for well-formedness, which we have addressed here.

7 Timeline of Proposed Accomplishments

This thesis proposes three important contributions in the realm of feature modularity in mechanized theorem proving:

1. I will show how the mechanized syntax, semantics and meta-theory proofs of a programming language can be effectively modularized into components that can be composed in different ways to build programming languages with fully mechanized meta-theory.
   Framework for composing reusable language modules. [Accomplished].
   FGJ and miniML case studies showing applicability of approach. [Accomplished].

2. I will demonstrate how modularization of semantic properties alongside definitions enables efficient reasoning about the entire family of products in both product lines of theorems and SPLs.
   Type system for checking safe composition of SPLs. [Accomplished].
   Techniques for efficiently checking safe composition of TPLs. [ETA: 3 months.]

3. I will investigate how these techniques can aid in the integration of interpreters for different languages, broadening its applicability through a monadic extension of MTC and through a study of automatic interaction detection for interpreter composition.
   Modular reasoning techniques for monadic interpreters. [ETA: 3 months]
   Investigation of feature interaction detection for interpreter composition. [ETA: 4 months]

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

Meta-Theory à la Carte

Benjamin Delaware  
University of Texas at Austin  
bendy@cs.utexas.edu

Bruno C. d. S. Oliveira  
National University of Singapore  
oliveira@comp.nus.edu.sg

Tom Schrijvers  
Universiteit Gent  
tom.schrijvers@ugent.be

Abstract

Formalizing meta-theory, or proofs about programming languages, in a proof assistant has many well-known benefits. Unfortunately, the considerable effort involved in mechanizing proofs has prevented it from becoming standard practice. This cost can be amortized by reusing as much of existing mechanized formalizations as possible when building a new language or extending an existing one. One important challenge in achieving reuse is that the inductive definitions and proofs used in these formalizations are closed to extension. This forces language designers to cut and paste existing definitions and proofs in an ad-hoc manner and to expend considerable effort to patch up the results.

The key contribution of this paper is the development of an induction technique for extensible Church encodings using a novel reinterpretation of the universal property of folds. These encodings provide the foundation for a framework, formalized in Coq, which uses type classes to automate the composition of proofs from modular components. This framework enables a more structured approach to the reuse of meta-theory formalizations through the composition of modular inductive definitions and proofs.

Several interesting language features, including binders and general recursion, illustrate the capabilities of our framework. We reuse these features to build fully mechanized definitions and proofs for a number of languages, including a version of mini-ML. Bounded induction enables proofs of properties for non-inductive semantic functions, and mediating type classes enable proof adaptation for more feature-rich languages.

Categories and Subject Descriptors D.3.1 [Programming Languages]: Formal Definitions and Theory—Semantics

Keywords Modular Mechanized Meta-Theory, Extensible Church Encodings, Coq

1. Introduction

With their POPLMARK challenge, Aydemir et al. [3] identified representation of binders, complex inductions, experimentation, and reuse of components as key challenges in mechanizing programming language meta-theory. While progress has been made, for example on the representation of binders, it is still difficult to reuse components, including language definitions and proofs.

The current approach to reuse still involves copying an existing formalization and adapting it manually to incorporate new features. An extreme case of this copy-&-adapt approach can be found in Leroy’s three person-year verified compiler project [22], which consists of eight intermediate languages in addition to the source and target languages, many of which are minor variations of each other. Due to the crosscutting impact of new features, the adaptation of existing features is unnecessarily labor-intensive. Moreover, from a software/formalization management perspective a proliferation of copies is obviously a nightmare. Typical formalizations present two important challenges to providing reuse:

1. Extensibility: Conventional inductive definitions and proofs are closed to extension and cannot simply be imported and extended with new constructors and cases. This is a manifestation of the well-known Expression Problem (EP) [45].

2. Modular reasoning: Reasoning with modular definitions requires reasoning about partial definitions and composing partial proofs to build a complete proof. However, conventional induction principles which are the fundamental reasoning techniques in most theorem provers only work for complete definitions.

The lack of reuse in formalizations is somewhat surprising, because proof assistants such as Coq and Agda have powerful modularity constructs including modules [26], type classes [17, 42, 46] and expressive forms of dependent types [11, 34]. It is reasonable to wonder whether these language constructs can help to achieve better reuse. After all, there has been a lot of progress in addressing extensibility [14, 30, 32, 43] issues in general-purpose languages using advanced type system features – although not a lot of attention has been paid to modular reasoning.

This paper presents MTC, a framework for defining and reasoning about extensible inductive datatypes. The framework is implemented as a Coq library which enables modular mechanized meta-theory by allowing language features to be defined as reusable components. Using MTC, language developers can leverage existing efforts to build new languages by developing new and interesting features and combining them with previously written components.

The solution to extensibility in MTC was partly inspired by the popular “Data types à la Carte” (DTC) [43] technique. However, DTC fundamentally relies on a type-level fixpoint definition for building modular data types, which cannot be encoded in Coq. MTC solves this problem by using (extensible) Church encodings of data types [30, 34, 35]. These encodings allow DTC-style modular data types to be defined in the restricted Coq setting. Another difference between DTC and MTC is the use of Mendler-style folds and algebras instead of conventional folds to express modular definitions. The advantage of Mendler-style folds [44] and algebras is that they offer explicit control over the evaluation order, which is important when modeling semantics of programming languages. MTC employs similar techniques to solve extensibility problems in proofs and inductively defined predicates.

MTC’s solution to modular reasoning uses a novel reinterpretation of the universal property of folds. Because MTC relies on
folds, the proof methods used in the initial algebra semantics of data types [16, 27] offer an alternative to structural induction. With some care, universal properties can be exploited to adapt these techniques to modular Church encodings. In addition to enabling modular reasoning about extensible inductive datatypes, universal properties also overcome some theoretical issues related to Church encodings in the Calculus of (Inductive) Constructions [34, 35].

MTC also supports ubiquitous higher-order language features such as binders and general recursion. Binders are modeled with a parametric HOAS [8] representation (a first-order representation would be possible too). Because these features require general recursion, they cannot be defined inductively using folds. To support these non-inductive features MTC uses a variation of mixins [10]. Mixins are closely related to Mendler-style folds, but they allow uses of general recursion, and can be modeled on top of Mendler-style Church encodings using a bounded fixpoint combinator.

To illustrate the utility of MTC, we present a case study modularizing several orthogonal features of a variant of mini-ML [9]. The case study illustrates how various features and partial type soundness proofs can be modularly developed and verified and later composed to assemble complete languages and proofs.

1.1 Contributions

The main contribution of our work is a novel approach to defining and reasoning about extensible inductive datatypes. MTC is a Coq framework for building reusable components that implements this approach to modular mechanized meta-theory.

More technically this paper makes the following contributions:

- **Extensibility Techniques for Mechanization**: The paper provides a solution to the EP and an approach to extensible mechanization of meta-theory in the restricted type-theoretic setting of Coq. This solution offers precise control over the evaluation order by means of Mendler folds and algebras. Mixins are used to capture ubiquitous higher-order features, like PHOAS binders and general recursion.

- **Non-Axiomatic Reasoning for Church Encodings**: The paper reinterprets the universal property of folds to recover induction principles for Mendler-style Church encodings. This allows us to avoid the axioms used in earlier approaches and preserves Coq’s strong normalization.

- **Modular Reasoning**: The paper presents modular reasoning techniques for modular components. It lifts the recovered induction principle from individual inductive features to compositions, while induction over a bounded step count enables modular reasoning about non-inductive higher-order features modeled with mixins.

MTC is implemented in the Coq proof assistant and the code is available at http://www.cs.utexas.edu/~bendy/MTC. Our implementation minimizes the user’s burden for adding new features by automating the boilerplate with type classes and default tactics. Moreover, the framework already provides modular components for mini-ML as a starting point for new language formalizations. We also provide a complimentary Haskell implementation of the computational subset of code used in this paper.

1.2 Code and Notational Conventions

While all the code underlying this paper has been developed in Coq, the paper adopts a terser syntax for its code fragments. For the computational parts, this syntax exactly coincides with Haskell syntax, while it is an extrapolation of Haskell syntax for proof-related concepts. The Coq code requires the impredicative-set option.

2. Extensible Semantics in MTC

This section shows MTC’s approach to extensible and modular semantic components in the restrictive setting of Coq. The approach is partly inspired by the DTC solution to the Expression Problem in Haskell, in particular its composition mechanisms for extensible inductive definitions. MTC differs from DTC in two important ways. Firstly, it uses Church encodings to avoid the termination issues of DTC’s generally recursive definitions. Secondly, it uses Mendler-style folds instead of conventional folds to provide explicit control over evaluation order.

2.1 Data Types à la Carte

This subsection reviews the core ideas of DTC. DTC represents the shape of a particular data type as a functor. That functor uses its type parameter \( a \) for inductive occurrences of the data type, leaving the data type definition open. \( \text{Arith}_F \) is an example functor for a simple arithmetic expression language with literals and addition.

\[
\text{data} \; \text{Arith}_F \; a = \text{Lit} \; \text{Nat} \mid \text{Add} \; a \; a
\]

The explicitly recursive definition \( \text{Fix}_{\text{DTC}} \; f \) closes the open recursion of a functor \( f \).

\[
\text{data} \; \text{Fix}_{\text{DTC}} \; f = \text{In} \; (f \; (\text{Fix}_{\text{DTC}} \; f))
\]

Applying \( \text{Fix}_{\text{DTC}} \) to \( \text{Arith}_F \) builds the data type for arithmetic expressions.

\[
\text{type} \; \text{Arith} = \text{Fix}_{\text{DTC}} \; \text{Arith}_F
\]

Functions over \( \text{Fix}_{\text{DTC}} \; f \) are expressed as folds of \( f \)-algebras.

\[
\text{type} \; \text{Algebra} \; f \; a = f \; a \rightarrow a
\]

\[
\text{fold}_{\text{DTC}} :: \text{Functor} \; f \Rightarrow \text{Algebra} \; f \; a \rightarrow \text{Fix}_{\text{DTC}} \; f \rightarrow a
\]

\[
\text{fold}_{\text{DTC}} \; \text{alg} \; (\text{In} \; \text{fa}) = \text{alg} \; (\text{fmap} \; \text{fold}_{\text{DTC}} \; \text{alg} \; \text{fa})
\]

For example, the evaluation algebra of \( \text{Arith}_F \) is defined as:

\[
\text{data} \; \text{Value} = I \; \text{Int} \mid B \; \text{Bool}
\]

\[
\text{eval}_{\text{Arith}} :: \text{Algebra} \; \text{Arith}_F \; \text{Value}
\]

\[
\text{eval}_{\text{Arith}} \; (\text{Lit} \; n) = I \; n
\]

\[
\text{eval}_{\text{Arith}} \; (\text{Add} \; (I \; v_1) \; (I \; v_2)) = I \; (v_1 + v_2)
\]

Note that the recursive occurrences in \( \text{eval}_{\text{Arith}} \) are of the same type as the result type \( \text{Value} \).

In essence, folds process the recursive occurrences so that algebras only need to specify how to combine the values (for example \( v_1 \) and \( v_2 \)) resulting from evaluating the subterms. Finally, the overall evaluation function is:

\[
[ ] :: \text{Fix}_{\text{DTC}} \; \text{Arith}_F \rightarrow \text{Value}
\]

\[
[ ] = \text{fold}_{\text{DTC}} \; \text{eval}_{\text{Arith}}
\]

\[
> ([(\text{Add} \; (\text{Lit} \; 1) \; (\text{Lit} \; 2))])
\]

Unfortunately, DTC’s two uses of general recursion are not permitted in Coq. Coq does not accept the type-level fixpoint combinator \( \text{Fix}_{\text{DTC}} \; f \) because it is not strictly positive. Coq similarly disallows the \( \text{fold}_{\text{DTC}} \) function because it is not structurally recursive.

2.2 Recursion-Free Church Encodings

MTC encodes data types and folds with Church encodings [6, 35], which are recursion-free. Church encodings represent (least) fixpoints and folds as follows:

\[
\text{type} \; \text{Fix} = \forall a. \text{Algebra} \; f \; a \rightarrow a
\]

\[
\text{fold} :: \text{Algebra} \; f \; a \rightarrow \text{Fix} \; f \rightarrow a
\]

\[
\text{fold} \; \text{alg} = \text{fa} \; \text{alg}
\]

\[1\] Boolean values are not needed yet, but they are used later in this section.
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 ArithF’s literals and addition are given by the lit and add functions:

\[
\begin{align*}
\text{lit} &::= \text{Nat} \rightarrow \text{Fix Arith}_F \\
\text{lit} \ n &:= \text{A}l\text{g} \rightarrow \text{alg} \ (\text{Lit} \ n) \\
\text{add} &::= \text{Fix Arith}_F \rightarrow \text{Fix Arith}_F \rightarrow \text{Fix Arith}_F \\
\text{add} \ e_1 \ e_2 &:= \text{A}l\text{g} \rightarrow \text{alg} \ (\text{Add} \ (\text{fold} \ \text{alg} \ e_1) \ (\text{fold} \ \text{alg} \ e_2))
\end{align*}
\]

The evaluation algebra and evaluation function are defined as in DTC, and expressions are evaluated in much the same way.

2.3 Lack of Control over Evaluation

Folds are structurally recursive and therefore capture compositionality of definitions, a desirable property of semantics. A disadvantage of the standard fold encoding is that it does not provide the implementer of the algebra with explicit control of evaluation. The fold encoding reduces all subterms; the only freedom in the algebra is whether or not to use the result.

**Example: Modeling if expressions**

As a simple example that illustrates the issue of lack of control over evaluation consider modeling if expressions and their corresponding semantics. The big-step semantics of if expressions is:

\[
\begin{align*}
\text{If} \ e_1 \ e_2 \ e_3 &\sim \text{true} & \text{If} \ e_1 \ e_2 \ e_3 &\sim \text{false} & \text{If} \ e_1 \ e_2 \ e_3 &\sim \text{v}_3 \\
\text{if} \ e_1 \ e_2 \ e_3 &\sim \text{v}_2 & \text{else} \ v_1 &\sim \text{v}_2 & \text{or} \ v_1 &\sim \text{v}_3
\end{align*}
\]

Using our framework of Church encodings, we could create a modular feature for boolean expressions such as if expressions and boolean literals as follows:

\[
\begin{align*}
\text{data} \ \text{Logic}_F \ a \ a &::= \text{If} \ a \ a \ a | \text{BLit} \ \text{Bool} \ -- \text{Boolean functor} \\
\text{eval}_{\text{Logic}} ::= \text{Arith}_F \text{ Logic}_F \text{ Value} \\
\text{eval}_{\text{Logic}} \ (\text{If} \ v_1 \ v_2 \ v_3) &:= \text{if} \ (v_1 \equiv B \ True) \text{ then} \ v_2 \ \text{else} \ v_3 \\
\text{eval}_{\text{Logic}} \ (\text{BLit} \ b) &:= B \ b
\end{align*}
\]

However, an important difference with the big-step semantics above is that eval_{Logic} cannot control where evaluation happens. All it has in hand are the values v₁, v₂ and v₃ that result from evaluation. While this difference is not important for simple features like arithmetic expressions, it does matter for if expressions.

Semantics guides the development of implementations. Accordingly, we believe that it is important that a semantic specification does not rely on a particular evaluation strategy (such as laziness). This definition of eval_{Logic} might be reasonable in a lazy meta-language like Haskell (which is the language used by DTC), but it is misleading when used as a basis for an implementation in a strict language like ML. In a strict language eval_{Logic} is clearly not a desirable definition because it evaluates both branches of the if expression. Aside from the obvious performance drawbacks, this is the wrong thing to do if the object language features, for example, non-termination. Furthermore, this approach can be quite brittle: in more complex object languages using folds and laziness can lead to subtle semantic issues [4].

2.4 Menderl-style Church Encodings

To express semantics in a way that allows explicit control over evaluation and does not rely on the evaluation semantics of the meta-language, MTC adapts Church encodings to use Menderl-style algebras and folds [44] which make recursive calls explicit.

**type** Algebra_M f a = ∀r.(r → a) → f r → a

A Menderl-style algebra differs from a traditional f-algebra in that it takes an additional argument (r → 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 Algebra_M f a, Menderl-style folds and Menderl-style Church encodings are defined as follows:

\[
\begin{align*}
\text{type} \ Fix_M f &:= \forall a. \text{Algebra}_M f a \rightarrow a \\
\text{fold}_M ::= \text{Algebra}_M f a \rightarrow \text{Fix}_M f a \\
\text{fold}_M \ f \ a &:= f \ a \ \text{alg}
\end{align*}
\]

Menderl-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 Menderl-style algebra is:

\[
\begin{align*}
\text{eval}_{\text{Logic}} ::= \text{Algebra}_M \text{ Logic}_F \text{ Value} \\
\text{eval}_{\text{Logic}} \ (\text{If} \ e_1 \ e_2 \ e_3) &:= \text{if} \ (e_1 \equiv B \ True) \text{ then} \ e_2 \ \text{else} \ e_3
\end{align*}
\]

Note that this definition allows explicit control over the evaluation order just like the big-step semantics definition. Furthermore, like the fold-definition, eval_{Logic} enforces compositionality because all the algebra can do to e₁, e₂ or e₃ is to apply the recursive call [ ].

2.5 A Compositional Framework for Menderl-style Algebras

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

**Modular Functors**

Individual features can be modularly defined using functors, like Arith_F and Logic_F. Functors are composed with the ⊕ operator:

\[
\begin{align*}
\text{data} \ (\oplus) \ g \ a &::= \text{Inl} \ (f \ a) | \text{Inr} \ (g \ a) \\
\text{Fix}_M \ (\text{Arith}_F \oplus \text{Logic}_F) &\text{ represents a data type isomorphic to:} \\
\text{data} \ Exp = \text{Lit} \ \text{Nat} | \text{Add} \ \text{Exp} \ Exp | \text{If} \ \text{Exp} \ \text{Exp} \ | \text{BLit} \ \text{Bool}
\end{align*}
\]

**Modular Menderl Algebras**

A type class is defined for every semantic function. For example, the evaluation function has the following class:

\[
\begin{align*}
\text{class} \ Eval \ f \ \text{where} \ \text{eval}_{alg} ::= \text{Algebra}_M f \text{ Value} \\
\text{In this class eval}_{alg} \text{ represents the evaluation algebra of a feature } f.
\end{align*}
\]

Algebras for composite functor are built from feature algebras:

\[
\begin{align*}
\text{instance} \ (\text{Eval} \ f, \text{Eval} \ g) &\Rightarrow \text{Eval} \ (f \oplus g) \ \text{where} \\
\text{eval}_{alg} \ (\text{Inl} \ \text{fexp}) &:= \text{eval}_{alg} \ (\text{fexp}) \\
\text{eval}_{alg} \ (\text{Inr} \ \text{gexp}) &:= \text{eval}_{alg} \ (\text{gexp})
\end{align*}
\]

Over all evaluation can then be defined as:

\[
\begin{align*}
\text{eval} ::= \text{Eval} f \Rightarrow \text{Fix}_M f \rightarrow \text{Value} \\
\text{eval} &:= \text{fold}_M \text{ eval}_{alg}
\end{align*}
\]

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 3 and subsumes all other algebra classes found in this paper. The paper continues to use more specific classes to make a gentler progression for the reader.

**Injections and Projections of Functors**

Figure 1 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
We can pattern match on particular features using expressions. Expressions are built with the smart constructors and used by operating instances. Hence, highly overlapping definitions like the first 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 [42] perform a backtracking search for matching instances. Hence, 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 \( \text{inj} \) function builds a new term from the application of \( \text{f} \) to some subterms.

\[
\text{inj} f :: \text{f} (\text{FixM} f) \rightarrow \text{FixM} f
\]

Smart constructors are built using \( \text{inj} \) and \( \text{inj} \) as follows:

\[
\text{inject} :: (g \ll f) \Rightarrow g (\text{FixM} f) \rightarrow \text{FixM} f
\]

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

\[
ex :: \text{FixM} (\text{ArithF} \oplus \text{LogicF})
\]

\[
\text{out}_f \text{ fexp} = \lambda \text{alg} \rightarrow \text{alg} (\text{foldM}_{\text{FixM} f} \text{fexp})
\]

The \( \text{out}_f \) function exposes the toplevel functor again:

\[
\text{out}_f :: \text{Functor f} \Rightarrow \text{FixM} f \rightarrow f (\text{FixM} f)
\]

We can pattern match on particular features using \( \text{prj} \) and \( \text{out}_f \):

\[
\text{isLit} :: (\text{ArithF} \ll f, \text{Functor f}) \Rightarrow \text{FixM} f \rightarrow \text{Maybe Nat}
\]

\[
\text{Just (Lit n)} \rightarrow \text{Just n}
\]

\[
\text{Nothing} \rightarrow \text{Nothing}
\]

2.6 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 \( \text{ArithF} \) and \( \text{LogicF} \) features. To easily achieve this extensibility, we make use of the same sorts of extensional encodings as the expression language itself:

\[
\text{data} NValF a = \text{I Nat}
\]

\[
\text{data} BValF a = \text{B Bool}
\]

\[
\text{data} \text{StuckF} a = \text{Stuck}
\]

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

To allow for an extensible return type \( r \) for evaluation, we need to parametrize the \( \text{Eval} \) type class in \( r \):

\[
\text{class} \text{Eval} f r \text{ where}
\]

\[
\text{eval} f r \text{ where}
\]

\[
\text{eval}_{\text{alg}} :: \text{Algebra}_{\text{M}} f (\text{FixM} r)
\]

Projection is now essential for pattern matching on values:

\[
\text{instance} (\text{StuckF} \ll r, \text{NValF} \ll r, \text{Functor r}) \Rightarrow
\]

\[
\text{Eval} \text{ArithF} r \text{ where}
\]

\[
\text{eval}_{\text{alg}} [\text{l}] (\text{Lit n}) = \text{vi n}
\]

\[
\text{case} \text{proj} [\text{e}] (\text{Add e1 e2}) =
\]

\[
\text{Just (I n1)} :: (\text{Just (I n2)}) \rightarrow \text{vi} (n1 + n2)
\]

\[
\text{Just (I n1), (Just (I n2))} \rightarrow \text{vi} (n1 + n2)
\]

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.

3. 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 [19].

3.1 The Problem of Church Encodings and Induction

Coq’s own original approach [35] 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 \( A_{\text{ind}} \), the induction principle for arithmetic expressions.

\[
A_{\text{ind}} :: \forall P :: (\text{Arith} \rightarrow \text{Prop}).
\]

\[
\forall \text{H}_i :: (\forall a. P \text{ Lit n})
\]

\[
\forall a. P a
\]

\[
A_{\text{ind}} P \text{ H}_i \text{ H}_a \text{ e} =
\]

\[
\text{case} \text{e} \text{ of} \text{ Lit n} = \text{H}_i \text{ n}
\]

\[
\text{Add x y} \rightarrow A_{\text{ind}} P \text{ H}_i \text{ H}_a \text{ x}
\]

\[
(A_{\text{ind}} P \text{ H}_i \text{ H}_a \text{ y})
\]
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 [34] 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. Hence, 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.

$$\forall e. \Gamma \vdash e : t \rightarrow \Gamma \vdash [e] : t$$

This restriction is a showstopper for the semantics setting of this paper, as it rules out proofs for most (if not all) theorems of interest. Supporting reasoning about semantic functions requires a new approach that does not suffer from this restriction.

### 3.2 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 P e holds for the new term. As the latter depends on the former, the result type of the algebra is a dependent product \( \Sigma e. P e \). A generic algebra can exploit this expressivity to build a poor-man’s induction principle, e.g., for the Arith F functor:

\[
A_{\text{ind}}^0 \::\! \forall P \::\! (\text{Fix}_{\text{M}} \text{Arith}_F \rightarrow \text{Prop}). \\
\forall H_1 \::\! (\forall n. P \langle \text{lit} n \rangle), \\
\forall H_2 \::\! (\forall a b. P a \rightarrow P b \rightarrow P (\text{add} a b)), \\
\text{Algebra} \text{Arith}_F (\Sigma e. P e) \\
A_{\text{ind}}^2 P H_1 H_2 e = \\
\text{case } e \text{ of} \\
\text{Lit } n \rightarrow \exists (\text{lit} n) (H_1 n) \\
\text{Add } x y \rightarrow \exists (\text{add} (\langle \text{pi} 1 x \rangle (\text{pi} 1 y)) (H_2 (\langle \text{pi} 1 x \rangle (\langle \text{pi} 1 y) (\langle \text{pi} 2 x \rangle (\langle \text{pi} 2 y)) \\
\text{Provided with the necessary proof cases, } A_{\text{ind}}^2 \text{ can build a specific proof algebra. The corresponding proof is simply a fold over a Church encoding using this proof algebra.}

Note that since a proof is not a computational object, it makes more sense to use regular algebras than Mendler algebras. Fortunately, regular algebras are compatible with Mendler-based Church encodings as the following variation of fold'_M shows.

\[
\text{fold}'_M :: \text{Functor } f \Rightarrow \text{Algebra } f a \rightarrow \text{Fix}_{\text{M}} f \rightarrow a \\
\text{fold}'_M \text{alg} = \text{fold}_M (\text{xrec} \rightarrow \text{alg} \circ \text{fmap rec})
\]

### 3.3 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 \( \text{Fix}_{\text{M}} f \) for Church encodings and \( \text{Algebra } f (\Sigma e. P e) \) for proof algebras. Statically ensuring this equivalence requires additional well-formedness conditions on both. These conditions formally capture our notion of Church encodings and proofs algebras.

#### 3.3.1 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.

\[
\forall \text{alg} :: \text{Algebra } f (\Sigma e. P e), \pi_1 \circ \text{alg} = \mu f \circ \text{fmap} \pi_1
\]

This constraint is encoded in the typeclass for proof algebras, PAlg. It is easy to verify that \( A_{\text{ind}}^0 \) satisfies this property. Other proof algebras over Arith _F can be defined by instantiating \( A_{\text{ind}}^0 \) with appropriate cases for \( H_1 \) and \( H_2 \). In general, well-formedness needs to be proven only once for any data type and induction algebra.

#### 3.3.2 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 fold'_M of it. So the fold'_M used to build a proof must be a proper fold'_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:

\[
h = \text{fold}_M \text{alg} \Leftrightarrow h \circ \text{in}_f = \text{alg} h
\]

In an initial algebra representation of an inductive data type, there is a single implementation of fold'_M that can be checked once and for all for the universal property. In MTC’s Church-encoding approach, every term of type \( \text{Fix}_{\text{M}} f \) consists of a separate fold'_M implementation that must satisfy the universal property. Note that this definition of the universal property is for a fold'_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 [18] uses the same characterization for deriving Church numerals.

Fortunately, the left-to-right implication follows trivially from the definitions of fold'_M and \( \text{in}_f \), independent of the particular term of type \( \text{Fix}_{\text{M}} f \). 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.

\[
\text{type UP } f e = \\
\forall a (\text{alg} :: \text{Algebra}_{\text{M}} f a) (h :: \text{Fix}_{\text{M}} f \rightarrow a). \\
(\forall e. h (\text{in}_f e') = \text{alg} h e') \Rightarrow h e = \text{fold}'_M \text{alg} e
\]

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 \( \text{in}_f \), that only builds terms with the universal property:

\[
\text{in}'_f :: \text{Functor } f \Rightarrow f (\Sigma e. \text{UP } f e) \rightarrow \Sigma e. \text{UP } f e
\]

about Church-encoded terms built from these smart-er constructors, as well as 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,

\[
h \circ \text{alg}_1 = \text{alg}_2 \circ \text{fmap} h \Rightarrow h \circ \text{fold}'_M \text{alg}_1 = \text{fold}'_M \text{alg}_2
\]

and the lesser known reflection law,

\[
\text{fold}'_M \text{in}_f = \text{id}
\]

#### 3.3.3 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 3.1. Given a functor \( f \), property \( P \), and a well-formed \( P \)-proof algebra \( \text{alg} \), for any Church-encoded \( f \)-term \( e \) with the universal property, we can conclude that \( P \ e \) holds.

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

\[
\begin{align*}
P \ (\pi_1 (\text{fold}_M' \ \text{alg} \ e)) & 
\implies \{-\text{well-founded algebra and fusion law } \} \\
P \ (\text{fold}_M' \ \text{alg} \ e) & 
\implies \{-\text{reflection law } \} \\
P \ e & 
\square
\end{align*}
\]

Theorem 3.1 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. Modular Proofs for Extensible Church Encodings

The aim of modularity in this setting is to first write a separate proof for every feature and then compose the individual proofs 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, since proofs are essentially folds of proof algebras, all of the reuse tools developed in Section 2 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.

4.1 Algebra Delegation

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

\[
A^\text{nd} = \forall f . \text{Arith}_F \triangleleft f \Rightarrow \\
\forall P :: (\text{Fix}_M f \to \text{Prop}). \\
\forall H_i :: (\forall n . P \ (\text{lit} n)). \\
\forall H_a :: (\forall a . P \ a \to P \ (\text{add} \ a \ b)). \\
\text{Algebra} \text{Arith}_F (\Sigma \ c . P \ c)
\]

Consider building a proof of

\[
\forall e . \text{typeof} \ e = \text{Just} \ \text{nat} \to \exists m :: \text{nat} . \text{eval} \ e = \text{vi} \ m
\]

using \( A^\text{nd} \). Then, the first proof obligation is

\[
\text{typeof} \ (\text{lit} \ n) = \text{Just} \ \text{nat} \to \exists m :: \text{nat} . \text{eval} \ (\text{lit} \ n) = \text{vi} \ m
\]

While this appears to follow immediately from the definition of \( \text{eval} \), recall that \( \text{eval} \) 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 \( \text{Arith}_F \)-algebra as expected:

\[
\forall r . (\text{rec} :: r \to \text{Nat}) . \text{eval}_\text{alg} \text{rec} \circ \text{inj} = \text{eval}_\text{alg} \text{rec}
\]

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.

```
class (Eval f, Eval g, f \triangleleft< g) \Rightarrow \\
WF.Eval f g \text{ where}
\text{wf.eval}_\text{alg} :: \forall (\text{rec} :: r \to \text{Nat}) (e :: f \ r). \\
\text{eval}_\text{alg} \text{rec} \ (\text{inj} \ e :: g \ r) = \\
\text{eval}_\text{alg} \text{rec} \ e
```

Figure 2. \( \text{WF.Eval} \) instances.

MTC provides the three instances of this class in Figure 2, one for each instance of \( \triangleleft< \) allowing Coq to automatically build a proof of well-formedness for every composite algebra.

4.1.1 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 \( \text{WF.Eval} \). These include the \( \text{DistinctSubFunctor} \) class, which ensures that injections from two different subfunctors are distinct, and the \( \text{WF.Functor} \) class that ensures that \( \text{fmap} \) distributes through injection.

Figure 3 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 2.

4.2 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 \( \text{Prop} \). For instance, a soundness proof makes use of a judgment about the well-typing of values.

```
data WTV :: Value \to Type \to Prop where
WTVat n : WTVat (i n) T Nat
WTVBool v :: Vab.WTVat (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., \( WTVat \) is indexed by a value and a type. This requires functions indexed by values \( x \) of type \( i \). For example, \( WTVat v t \) is the corresponding indexed functor for the extensible variant of \( WTVat \) above.

```
data WTVat :: v \to t \to (WTV :: (v, t) \to Prop) \to (v, t) \to Prop
where WTVat :: \forall n . (VNatF \triangleleft< v, Functor v, 
\forall n . (NValF \triangleleft< t, Functor t) 
\Rightarrow WTVat v t TV (n, n, nval)
```

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 \( WTVat \) functor.

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

```
type iAlg i f :: (i \to Prop) \to (i \to Prop) \ a 
= \forall x :: i . f \ a x \to a x

type iFix i f :: (i \to Prop) \to (i \to Prop) \ (x :: i) 
= \forall a :: i . \text{Prop} . iAlg f a \to a x ...
```

As these indexed variants are meant to construct logical relations, their parameters range over \( \text{Prop} \) instead of \( \text{Set} \). Fortunately,
Figure 3. Type classes provided by MTC

<table>
<thead>
<tr>
<th>Class Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>class Functor f where fnop :: (a -&gt; b) -&gt; (f a -&gt; f b) fnop_id :: fnop id = id fnop_fusion :: Vg h, fnop h o fnop g = fnop (h o g)</td>
<td>Functors Supplied by the user</td>
</tr>
<tr>
<td>class f &lt;~&gt; g where inj :: f a -&gt; g a prj :: g a -&gt; Maybe (f a) inj_prj :: prj ga = Just fa -&gt; ga = inj fa prj_inj :: prj o inj = Just</td>
<td>Functor Subtyping Inferred</td>
</tr>
<tr>
<td>class (Functor f, Functor g, f &lt;~&gt; g) =&gt; WF_Functor f g where approx :: ~a b (h :: a -&gt; b). fnop h o inj = inj o fnop h</td>
<td>Functor Delegation Inferred</td>
</tr>
<tr>
<td>class (Functor h, f &lt;<del>&gt; g, h &lt;</del>&gt; h) =&gt; DistinctSubFunctor f g h where inj_discriminate :: ~a b (f :: a) (ge :: g a), inj fe /= inj ge</td>
<td>Functor Discrimination Inferred</td>
</tr>
<tr>
<td>class (Functor f, Functor g, f &lt;~&gt; g) =&gt; FAlg name t a f where f_algebra :: Mixin t f a</td>
<td>Function Algebras Supplied by the user</td>
</tr>
<tr>
<td>class (f &lt;~&gt; g, FAlg n t a f, FAlg n t a g) =&gt; WF_FAlg n t a f g where approx :: ~a b (h :: a -&gt; b). fnop h o inj = inj o fnop h</td>
<td>Algebra Delegation Inferred</td>
</tr>
<tr>
<td>class (Functor f, Functor g, f &lt;~&gt; g) =&gt; PAlg name f g a where p_algebra :: Algebra f a proj_eq :: ~e, m, p_algebra e = inj f (inj (fmap p) e))</td>
<td>Proof Algebras Supplied by the User</td>
</tr>
</tbody>
</table>

4.3 Case Study: Soundness of an Arithmetic Language

Here we briefly illustrate modular reasoning with a case study proving soundness for the Arith_F :: Logic_F language.

The previously defined eval function captures the operational semantics of this language in a modular way and reduces an expression to a NVal_F :: BVal_F :: Stuck_F value. Its type system is similarly captured by a modularly defined type-checking function typeof that maybe returns a NVal_F :: BBool value. In lieu of FixM in function signatures:

\[ \text{typeof } \]

data TNat_F t = TNat
data TBool_F t = TBool

For this language soundness is formulated as:

**Theorem soundness ::**

\[ \forall e \in \text{env}, \text{typeof } e = \text{Just } t \rightarrow \text{WTValue } (\text{eval } e \text{ env}) t \]

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 superfunction 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 Logic_F case of the soundness property: given that typeof (if c e1 e2) = Some t1, we wish to show that WTValue (eval (if c e1 e2)) t1. If c evaluates to false, we need to show that WTValue e2 t1.

Since if c e1 e2 has type t1, the definition of typeof says that e1 has type t1,

\[ \text{typeof}_\text{alg} \text{ rec } (\text{if } c \text{ e1 e2}) = \]

\[ \text{case } \text{project } (\text{rec } c) \text{ of } \]

\[ \text{Just } \text{TBool } \rightarrow \]

\[ \text{case } (\text{rec } e1, \text{rec } e2) \text{ of } \]

\[ \text{Just } t1, \text{Just } t2 \rightarrow \]

\[ \text{if } \text{eqtype } t1 t2 \text{ then } t1 \text{ else Nothing } \]

\[ \rightarrow \text{Nothing} \]

\[ \rightarrow \text{Nothing} \]

In addition, the type equality test function, eq_tag, says that e1 and e2 have the same type: eq_tag t1 t2 = true. We need to make use of a sublemma showing that \( \forall t, t_t, t_{-1} t_2 = t_2 \). We have seen, in order to do so, the universal property must hold for typeof e1. This is easily accomplished by packaging a proof of the universal property alongside t1 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: lit j = lit k \( \leftrightarrow j = k \), 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 inj_f. To facilitate this packaging, we provide a type synonym that can be used in lieu of FixM.

\[ \text{type } \text{UP}_F f = \text{Functor } f \Rightarrow \Sigma e, (\text{UP } f e) \]

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, \( \text{UP}_F \), that captures this and augments induction principles accordingly.

\[ \text{UP}_P :: \text{Functor } f \Rightarrow (P \forall e, \text{UP } f e \Rightarrow \text{Prop}) \Rightarrow (e :: \text{FixM } f) \rightarrow \Sigma e. (P e) \]

**Equality and Universal Properties** While packaging universal properties with terms enables reasoning, it does obfuscate equality of terms. In particular, two \( \text{UP}_F \) terms t and t' may share the same underlying term (i.e., \( \pi_1 t = \pi_1 t' \)), while their universal property proof components are different.

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

\[ ^2 \text{Actually, as proofs are opaque, we cannot tell if they are equal.} \]
5. Higher-Order Features

Binders and general recursion are ubiquitous in programming languages, so MTC must support these sorts of higher-order features. The untyped lambda calculus demonstrates the challenges of implementing both these features with extensible Church encodings.

5.1 Encoding Binders

To encode binders we use a parametric HOAS (PHOAS) [8] representation. PHOAS allows binders to be expressed as functors, while still preserving all the convenient properties of HOAS.

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

data LambdaF v r = Var v | App r r | Lam (r -> r)

As before, smart constructors build extensible expressions:

var :: (LambdaF v <:: f) => v -> FixMF
var v = inject (Var v)

app :: (LambdaF v <:: f) => FixMF -> FixMF -> FixMF
app e1 e2 = inject (App e1 e2)

lam :: (LambdaF v <:: f) => (v -> FixMF) -> FixMF
lam f = inject (Lam f)

5.2 Defining Non-Inductive Evaluation Algebras

Defining an evaluation algebra for the LambdaF feature presents additional challenges. Evaluation of the untyped lambda calculi presents a closure, requiring a richer value type than before:

data Value =
Stuck | J Nat | B Bool | Clos (Value -> Value)

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 ClosureF integrates closure values into the framework of extensible values introduced in Section 2.6.

data ClosureF f a = Clos (FixMF f) (Env a)
closure :: (ClosureF f <:: r) => FixMF f -> Env (FixMF r) -> FixMF r
closure mf e = inject (Clos mf e)

A first attempt at defining evaluation is:

evalLambda :: (ClosureF f <:: r, StuckF <:: r, Functor r) => AlgebraF (LambdaF Nat) (Env (FixMF r) -> FixMF r)
evalLambda [ ] exp env =
  case exp of
  Var index -> env !! index

The function evalLambda instantiates the type variable v of the LambdaF 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 evalLambda simply looks up that index in the environment. In the lambda case evalLambda builds a closure using f and the environment. Finally, in the application case, the expression e1 is evaluated and analyzed. If that expression evaluates to a closure then the expression e2 is evaluated and added to the closure’s environment (env’), and the closure’s expression e3 is evaluated under this extended environment. Otherwise e1 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 Nat -> FixMF f. Instead, its type is Nat -> r, where r is universally quantified in the definition of the AlgebraF feature. Secondly, and symmetrically, in the App case, the closure expression e3 has type FixMF f which does not conform to the type r expected by [] 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 FixMF 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.

5.3 Non-Inductive Semantic Functions

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

type Mixin t f a = (t -> a) -> t -> a

This algebra specifies the type t of subterms, typically FixMF f, the overall expression type. With this mixin algebra, evalLambda is now well-typed:

evalLambda :: (ClosureF e <:: v, StuckF <:: v) => Mixin (FixMF e) (LambdaF Nat)
  (Env (FixMF e) -> FixMF e)

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

class EvalX f g r where
evalalg :: Mixin (FixMF f) g (Env (FixMF r) -> FixMF r)
instance (StuckF <:: r, ClosureF <:: r, Functor r) =>
  EvalX f (LambdaF Nat) r where
evalalg = evalLambda

Although the code of evalLambda still appears generally recursive, it is actually not because the recursive calls are abstracted as a parameter (like with Mendler algebras). Accordingly, evalLambda 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 [10]. 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, \textit{boundedFix}, that returns a default value if the evaluation does not terminate after \( n \) recursion steps.

\[
\text{boundedFix} :: \forall f . \text{Functor } f \Rightarrow \text{Nat} \rightarrow a \rightarrow \text{Mixin } (\text{Fix}_M f) a \rightarrow \text{Fix}_M f a
\]

\[
\text{boundedFix } n \text{ def } \text{alg } e = \begin{cases} \text{case } n \text{ of} \\ 0 & \rightarrow \text{def} \\ m & \rightarrow \text{alg} (\text{boundedFix} (m - 1) \text{ def } \text{alg}) (\text{out}_f e) \end{cases}
\]

The argument \( e \) is a Mendler-encoded expression of type \( \text{Fix}_M f \).

\text{boundedFix} first uses \text{out}_f to unfold the expression into a value of type \( f \text{ (Fix}_M f) \) and then applies the algebra to that value recursively. In essence \text{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 \( \text{Eval}_X \). The definition uses a distinguished bottom value, \( \bot \), that represents a computation which does not finish within the given bound.

\[
data \bot_F a = \text{Bot} \\
\bot = \text{inject } \text{Bot}
\]

\[
\text{eval}_X :: (\text{Functor } f, \bot_F \triangleleft r, \text{Eval}_X f f r) \Rightarrow \\
\text{Nat} \rightarrow \text{Fix}_M f \rightarrow \text{Env} \rightarrow \text{Fix}_M r
\]

\[
\text{eval}_X n e \text{ env} = \text{boundedFix } n (\bot, \bot) \text{ eval}_\text{alg} e \text{ env}
\]

### 5.4 Backwards compatibility

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

1. \( \text{eval}_X \) uses mixin algebras instead of Mendler algebras.
2. \( \text{eval}_X \) 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 \textit{mendlerToMixin} function automatically derives the required mixin algebra from the Mendler algebra.

\[
\text{mendlerToMixin} :: \text{Algebra}_M f a \rightarrow \text{Mixin } (\text{Fix}_M g) f a
\]

\[
\text{mendlerToMixin alg } = \text{alg}
\]

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

\[
\text{instance } \text{Eval Arith}_F f \Rightarrow \text{Eval}_X f \text{ Arith}_F r \text{ where } \\
\text{eval}_\text{alg} e \text{ env} = \\
\text{mendlerToMixin evalAlgebra } (\text{flip } \cdot \text{ env} e)
\]

The algebras of binder-free features can be similarly adapted to build an algebra over a parametric functor. Figure 4 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.

### 6. Reasoning with Higher-Order Features

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

**Theorem** \( \textit{soundness}_X :: \forall f . \text{Functor } f \Rightarrow \text{T} \leftrightarrow \text{Nat} \rightarrow \text{Fix}_M (f \text{ (Maybe } (\text{Fix}_M f)))). \)

\[
\forall e_1 :: \text{Fix}_M (f \text{ (Nat)}).
\]

\[
\forall e_2 :: \text{Fix}_M (f \text{ (Maybe } (\text{Fix}_M f))).
\]

**Figure 4. Hierarchy of Algebra Adaptation**

\[
\text{Gamma } \vdash e_1 \equiv e_2 \rightarrow \text{WF-Env} \rightarrow \text{Gamma } \text{ env } \rightarrow \text{Typeof } e_1 = \text{Just } t \rightarrow \text{WTV} \text{ (eval}_X n e_2 \text{ env) } t
\]

The proof of \( \textit{soundness}_X \) features two substantial changes to the proof of \( \textit{soundness} \) from Section 4.3.

### 6.1 Proofs over Parametric Church Encodings

The statement of \( \textit{soundness}_X \) uses two instances of the same PHOAS expression \( e :: \forall v . \text{Fix}_M (f v) \). 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 \( \textit{soundness}_X \), 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 [8] shows how the connection can be retained with the help of an auxiliary equivalence relation \( \Gamma \vdash e_1 \equiv e_2 \), which uses the environment \( \Gamma \) 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 \( \textit{soundness}_X \) can then analyze \( e_1 \) and \( e_2 \) in lockstep. Figure 5 shows the rules for determining equivalence of lambda expressions.

\[
\begin{align*}
(x, x') \in \Gamma & \quad \Gamma \vdash e_1 \equiv e_1' \quad \Gamma \vdash e_2 \equiv e_2' \\
\Gamma \vdash \text{app } e_1 \text{ env } e_2 \equiv \text{app } e_1' \text{ env } e_2' & \quad \text{(EQV-VAR)}
\end{align*}
\]

\[
\begin{align*}
\forall x, x' . (x, x') \in \Gamma & \quad \Gamma \vdash \text{app } e_1 \text{ env } e_2 \equiv \text{app } e_1' \text{ env } e_2' \\
\Gamma \vdash \text{lam } f \equiv \text{lam } f' & \quad \text{(EQV-ABS)}
\end{align*}
\]

**Figure 5. Lambda Equivalence Rules**

### 6.2 Proofs for Non-Inductive Semantics Functions

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

In the base case of \( \textit{soundness}_X \), the bound has been reached and \text{eval}_X returns \( \bot \). The proof of this case relies on adding to the \textit{WTV} judgment the WF-Bot rule stating that every type is inhabited by \( \bot \).

\[
\bot \vdash T \quad \text{(WF-BOT)}
\]

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: \( \text{IH} :: \forall n, e, (\forall e', P(f \text{ (fun } e')) \rightarrow P(f \text{ (n+1) fun } e)) \). The \text{App } e_1 \text{ env } e_2 case of the soundness theorem illustrates the reason for including \text{IH} in the statement of the proof algebra. After
using the induction hypothesis to show that eval\_x \ e1 \ env produces a well-formed closure Clos \ e1 \ env', we must then show that evaluating e3 under the (eval\_x \ e2 \ env) \ env' environment is also well-formed. However, e3 is not a subterm of App \ e1 \ e2, so the conventional induction hypothesis for subterms does not apply. Because eval\_x \ e3 ((eval\_x \ e2 \ env') \ env') is run with a smaller bound, the bounded induction hypothesis IH\_c can be used.

6.3 Proliferation of Proof Algebras

In order to incorporate non-parametric inductive features in the soundness\_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 \( \llbracket \cdot \rrbracket \)), a naive approach requires four different proof algebras for an inductive non-parametric feature.\(^3\) This is not acceptable, 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 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.

6.3.1 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 \( f_{np} \) can be defined generically:

\[
\Gamma \vdash \pi \equiv b \quad \text{(Eqv-NP)}
\]

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

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

\[
\text{instance (PAlg N P f}_{np} \Rightarrow iPAlg N P \text{ (Eqv-NP f}_{np})
\]

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

6.3.2 Adapting Proofs to Non-Inductive Semantic Functions

To be usable regardless of whether \( \text{fold}_M \) or \( \text{boundedFix} \) 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:

\[
\begin{align*}
\text{type soundness e t.p ev} = \\
\forall \text{env t.p (out}_f \text{ (}\pi_1 \text{ e)}) = \text{Just t} \rightarrow \text{WTV value (ev (out}_{f'} \text{ (}\pi_1 \text{ e)}) \text{ env t)}
\end{align*}
\]

The hypothesis \( IH\_c \) is used to relate calls of \( \text{rec}_e \) and \( \text{rec}_t \) to applications of \( \text{eval}_{alg} \) and \( \text{typeof}_{alg} \).

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 \( \text{boundedFix} \).

\[
\text{instance (PAlg N P E) \Rightarrow iPAlg N (IH \rightarrow P) E}
\]

7. Case Study

As a demonstration of the MTC framework, we have built a set of five reusable language features and combined them into a mini-ML [9] variant. The study also builds five other languages from these features.\(^4\) Figure 6 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 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.

![Figure 6. mini-ML expressions, values, and types](image)

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 [5], 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 \( \text{nat} \) are natural numbers: \( \lambda x : \text{nat} \rightarrow x :: \text{N} \). The \( \text{Bool} \) 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 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 the case study, feature interactions consist almost exclusively of inversion principles for judgments and the projection principles of Section 4.3. 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.

---

\(^3\) Introducing type-level binders would further complicate the situation with four possible signatures for the \( \text{typeof}_{alg} \).

\(^4\) Also available at [http://www.cs.utexas.edu/~bendy/MTC](http://www.cs.utexas.edu/~bendy/MTC)
8. Related Work

This section discusses related work.

Modular Reasoning There is little work on mechanizing modular proofs for extensible components. An important contribution of our work is how to use universal properties to provide modular reasoning techniques for encodings of inductive data types that are compatible with theorem provers like Coq. Old versions of Coq, based on the calculus of constructions [11], also use Church encodings to model inductive data types [35]. However, the inductive principles to reason about those encodings had to be axiomatized, which endangered strong normalization of the calculus. The calculus of inductive constructions [34] has inductive data types built-in and was introduced to avoid the problems with Church encodings. MTC returns to Church encodings to allow extensibility, but does not use standard, closed induction principles. It instead uses a reasoning framework based on universal properties which allow modular reasoning without axioms in Coq.

Extensibility Our approach to extensibility combines and extends ideas from existing solutions to the expression problem. The type class infrastructure for (Mendler-style) \( f \)-algebras is inspired by DTC [14, 43]. However the type-level fixpoints that are central to DTC cannot be used in Coq because of their use of general recursion. To avoid general recursion, MTC encodes least-fixpoints with Church encodings [6, 35]. Church encodings have inspired other solutions to the expression problem (especially in object-oriented languages) [30–32]. Those solutions do not use \( f \)-algebras: instead, they use an isomorphic representation called object algebras [31]. Object algebras are a better fit for languages where records are the main structuring construct (such as OO languages). MTC differs from previous approaches by using Mendler-style \( f \)-algebras instead of conventional \( f \)-algebras or object algebras. Unlike previous solutions to the expression problem, which focus only on the extensibility aspects of implementations, MTC also deals with modular reasoning and extensible inductively defined predicates.

Mechanized Meta-Theory and Reuse Several ad-hoc tool-based approaches provide reuse, but none is based on a proof assistant’s modularity features alone. The Tinkertype project [23] is a framework for modularly specifying formal languages. It was used to format the language variants used in Pierce’s “Types and Programming Languages” [37], and to compose traditional pen-and-paper proofs. The Ott tool [41] allows users to write definitions and theorems in an ASCII format designed to mirror pen-and-paper formalizations. These are then automatically translated to definitions in either Is\(\text{\footnotesize{TP}}\) or a theorem prover and proofs and functions are then written using the generated definitions.

Both Boïte [7] and Mulhern [29] consider how to extend existing inductive definitions and reuse related proofs in the Coq proof assistant. Both their techniques rely on external tools which are no longer available and have users write extensions with respect to an existing specification. As such, features cannot be checked independently or easily reused with new specifications. In contrast, our approach is fully implemented within Coq and allows for independent development and verification of features.

Delaware et al. [13] applied product-line techniques to modularizing mechanism meta-theory proofs. As a case study, they built type safety proofs for a family of extensions to Featherweight Java from a common base of features. Importantly, composition of these features was entirely manual, as opposed to the automated composition developed here.

Concurrently with our development of MTC, Schwaab et al. have been working on modularizing meta-theory in Agda [40]. While MTC uses Church encodings to encode extensible datatypes, their approach achieves extensibility by using universes which can be lifted to the type level. Encodings and their associated proofs can be modified to derive new languages.

Transparency One long-standing criticism of mechanized meta-theory has been that it interferes with adequacy, i.e. convincing users that the proven theorem is in fact the desired one [39]. Certainly the use of PHOAS can complicate the transparency of mechanized definitions. The soundness theorem, for example, uses a more complicated statement than the pen-and-paper version because PHOAS requires induction over the equivalence relation. 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 the composition mechanisms of MTC is an interesting direction for future work. Such a higher-level notation could help with transparency; while MTC’s composition mechanisms could help with generating modular code for Ot
definitions.

Binding To minimize the work involved in modeling binders, MTC provides reusable binder components. The problem of modeling binders has previously received a lot of attention. Some proof assistants and type theories address this problem with better support for names and abstract syntax [36, 38]. In general-purpose proof assistants like Coq, however, such support is not available.

A popular approach, widely used in Coq formalizations, is to use mechanization-friendly first-order representations of binders such as the locally nameless approach [1]. This involves developing a number of straightforward, but tedious infrastructure lemmas and definitions for each new language. Such tedious infrastructure can be automatically generated [2] or reused from data type-generic definitions [21]. However this typically requires additional tool support. A higher-order representation like PHOAS [8] avoids most infrastructure definitions. While we have developed PHOAS-based binders in MTC, it supports first-order representations as well.

Semantics and Interpreters While the majority of semantics formalization approaches use inductively defined predicates, we propose an approach based on interpreters. Of course, MTC supports standard approaches as well.

A particularly prominent line of work based on interpreters is that of using \(\text{monads}\) to structure semantics. Moggi [28] pioneered monads to model computation effects and structure denotation semantics. Liang et al. [25] introduced \(\text{monad transformers}\) to compose multiple monads and build modular interpreters. Jaskelioff et al. [20] used an approach similar to DTC in combination with monads to provide modular implementation of mathematical operational semantics. Our work could benefit from using monads to model more complex language features. However, unlike previous work, we also have to consider modular reasoning. Monads introduce important challenges in terms of modular reasoning. Only very recently have some modular proof techniques for reasoning about monads been introduced [15, 33]. While these are good starting points, it remains to be seen whether these techniques are sufficient to reason about suitably generalized modular statements like soundness.

Mechanization of interpreter-based semantics clearly poses its own challenges. Yet, it is highly relevant as it bestows the high degree of confidence in correctness directly on the executable artifact, rather than on an intermediate formulation based on inductively defined relations. The only similar work in this direction, developed concurrently to our own, is that of Danielsson [12]. He uses the \(\text{partiality monad}\), which fairly similar to our bounded fixpoint, to formalize semantic interpreters in Agda. He argues that this style is more easily understood and more obviously deterministic and
computable than logical relations. Unlike us, Danielsson does not consider modularization of definitions and proofs.

9. Conclusion
Formalizing meta-theory can be very tedious. For larger programming languages the required amount of work can be overwhelming.

We propose a new approach to formalizing meta-theory that allows modular development of language formalizations. By building on existing solutions to modularity problems in conventional programming languages, MTC allows modular definitions of language components. Furthermore, MTC supports modular reasoning about these components. Our approach enables reuse of modular inductive definitions and proofs that deal with standard language constructs, allowing language designers to focus on the interesting constructs of a language.

This paper addresses many, but obviously not all, of the fundamental issues for providing a formal approach to modular semantics. We will investigate further extensions of our approach, guided by the formalization of larger and more complex languages on top of our modular mini-ML variant. A particularly challenging issue we are currently considering is the pervasive impact of new side-effecting features on existing definitions and proofs. We believe that existing work on modular monadic semantics [20, 24, 25] is a good starting point to overcome this hurdle.

Acknowledgements
We would like to especially thank William Cook for his help in structuring the presentation of this work. Further thanks to Don Batory, Twan van Laarhoven and the reviewers for comments and suggestions. This work was supported by the National Science Foundation under Grant CCF 0724979.

References
Product Lines of Theorems

Benjamin Delaware    William R. 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 development can 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 product lines with theorems and proofs 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 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.

Categories and Subject Descriptors D.3.1 [Programming Languages]: D.3.1 Formal Definitions and Theory

General Terms Design, Theory, Verification

Keywords Feature-Orientation, Mechanized Metatheory, Product Line Verification

1. Introduction
Mechanized theorem proving is hard: large-scale proof developments [13, 16] 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 similar proofs. The problem is compounded when verifying members of a product line – a family of related systems [2, 5] – 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 for software product lines is that their contents cut across normal object-oriented boundaries [5, 25]. 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) [14], 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 type system statically enforces the desired run-time behavior of a language, typically preservation and progress [24].

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 as an example. We implement feature modules in Coq [8] 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 [18, 23]. So too has the challenge of achieving object-oriented code reuse in this context [26, 30]. 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 [7]. 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 [14] 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 right-hand column presents the corresponding definitions for FGJ.

Figure 1: Selected FJ Definitions with FGJ Changes Highlighted
<table>
<thead>
<tr>
<th>FJ Fields of a Supertype Lemma</th>
<th>FGJ Fields of a Supertype Lemma</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Lemma 2.1.</strong> If ( S \vdash T ) and ( \text{fields}(T) = T \text{f} ), then ( \text{fields}(S) = S \text{g} ) and ( S_i = T_i ) and ( g_i = f_i ) for all ( i \leq #(f) ).</td>
<td><strong>Lemma 2.2.</strong> If ( \Delta \vdash S \vdash T ) and ( \text{fields}(\text{bound}<em>{\Delta}(T)^g) = T ), then ( \text{fields}(\text{bound}</em>{\Delta}(S)^g) = S \text{g} ), ( S_i = T_i ) and ( g_i = f_i ) for all ( i \leq #(f) ).</td>
</tr>
</tbody>
</table>

**Proof.** By induction on the derivation of \( S \vdash T \) | **Proof.** By induction on the derivation of \( \Delta \vdash S \vdash T \)

 Case **S-REFL** \( S = T \). Follows immediately. | Case **GS-REFL** \( S = T \). Follows immediately. |

 Case **S-TRANS** \( S \vdash V \) and \( V \vdash T \). By the inductive hypothesis, \( \text{fields}(V) = V \text{f} \) and \( V_i = T_i \) and \( h_i = f_i \) for all \( i \leq \#(f) \). Again applying the inductive hypothesis, \( \text{fields}(S) = S \text{g} \) and \( S_i = V_i \) and \( g_i = h_i \) for all \( i \leq \#(h) \). Since \( \#(f) \leq \#(h) \), the conclusion is immediate. | Case **GS-TRANS** \( \Delta \vdash S \vdash V \) and \( \Delta \vdash V \vdash T \). By the inductive hypothesis, \( \text{fields}(\text{bound}_{\Delta}(V)^g) = V \text{f} \) and \( V_i = T_i \) and \( h_i = f_i \) for all \( i \leq \#(f) \). Again applying the inductive hypothesis, \( \text{fields}(\text{bound}_{\Delta}(S)^g) = S \text{g} \) and \( S_i = V_i \) and \( g_i = h_i \) for all \( i \leq \#(h) \). Since \( \#(f) \leq \#(h) \), the conclusion is immediate. |

 Case **S-DIR** \( S = C \), \( T = D \), class \( C \) extends \( D \{ S \text{g}; \ldots \} \). By the rule \( \text{F-CLASS} \), \( \text{fields}(C) = U.T; S \text{g} \), where \( U.T = \text{fields}(D) \), from which the conclusion is immediate. | Case **GS-DIR** \( S = C \{ T \}^\beta \), \( T = [T/X]^\beta \text{D} \{ V \}^\beta \), class \( C \{ X \sim N \}^\beta \) extends \( D \{ V \}^\beta \{ S \text{g}; \ldots \} \). By the rule \( \text{F-CLASS} \), \( \text{fields}(C \{ T \})^\beta = U.T \{ T/X \}^\beta \text{D} \{ V \}^\beta \text{Sg} \), where \( U.T = \text{fields}( [T/X]^\beta \text{D} \{ V \}^\beta ) \). By definition, \( \text{bound}_{\Delta}(V) = V \) for all non-variable types \( V^\gamma \), from which the conclusion is immediate. |

**Figure 2:** An Example FJ Proof with FGJ Changes Highlighted

\(\alpha\). 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.

\(\beta\). Modifying existing syntax. FGJ adds type parameters to method calls, object creation, casts, and class definitions.

\(\gamma\). 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.

\(\delta\). Extending judgment signatures. The added rule GS-VAR looks up the bound of a type variable using a typing context, \( \Delta \). This context must be added to the signature of the subtyping relation, transforming all occurrences to a new ternary relation.

\(\eta\). 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. 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. 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. 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 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 com-
mon 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 that are semantically correct.

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 within the Coq proof assistant. In our approach, each feature has a set of assumptions that serve as variation 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 [28] omit casts). There are also optional features, which are unary functions, that extend the base or other features:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cFJ</td>
<td>core Featherweight Java</td>
</tr>
<tr>
<td>Cast</td>
<td>adds casts to expressions</td>
</tr>
<tr>
<td>Interface</td>
<td>adds interfaces</td>
</tr>
<tr>
<td>Generic</td>
<td>adds type parameters</td>
</tr>
</tbody>
</table>

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 FJ
Cast · cFJ // Original FJ [14]
Interface · Cast · cFJ // FJ with Interfaces
Generic · Cast · cFJ // Core Featherweight Generic Java
Generic · Interface · cFJ // core Generic FJ with Interfaces
Generic · Interface · Cast · cFJ // Generic Interfaces
Generic · Interface · Cast · cFJ // Generic Interfaces
```

3.2 Feature Models

Not all compositions of features 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 [6]. The grammar of feature model P (below) defines eight sentences (features k, i, j are optional; b is mandatory). Constraints limit legal sentences to those that have at least one optional feature, and if feature k is selected, so too must j.

```
P : [k] [i] [j] b; // context free grammar
k ∨ j ∨ i; // additional constraints
k → j;
```

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

```
L : [Generic][Interface][Cast]cFJ;
```

3.3 Multiple Representations of Languages

Every base language (cFJ) has multiple representations: its syntax \(s_{cFJ}\), operational semantics \(o_{cFJ}\), type system \(t_{cFJ}\), and metatheory proofs \(p_{cFJ}\). A base language is a tuple of representations \(cFJ = [s_{cFJ}, o_{cFJ}, t_{cFJ}, p_{cFJ}]\). An optional feature \(i\) extends each representation: the language’s syntax is extended with new productions \(\Delta s_i\), its operational semantics are extended by modifying existing rules and adding new rules to handle the updated syntax \(\Delta o_i\), etc. Each of these changes is modeled by a unary function. Feature \(i\) is a tuple of such functions \(i = [\Delta s_i, \Delta o_i, \Delta t_i, \Delta p_i]\) that update each representation of a language.

The representations of a language are computed by composing tuples element-wise. The tuple for language FJ = Cast · cFJ is:

```
FJ = Cast · cFJ
= [ΔsC, ΔoC, ΔtC, ΔpC] · [sFJ, oFJ, tFJ, pFJ]
= [ΔsC · sFJ, ΔoC · oFJ, ΔtC · tFJ, ΔpC · pFJ]
```

That is, the syntax of FJ is the syntax of the base \(s_{FJ}\) composed with extension \(Δs_i\), the semantics of FJ are the base semantics \(o_{FJ}\) composed with extension \(Δo_i\), and so on. In this way, all parts of a language are updated lock-step when features are composed. See [5, 12] for generalizations of these ideas.

3.4 Feature Interactions

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

```
J ::= interface I {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:

\[ J ::= \text{interface I } (X \in \mathbb{N}) \{ \text{Mty} \} \]

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 additional 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 #). 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:

\[ f \times g = (f\#g) \cdot f \cdot g \tag{1} \]

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

\[ f\#(g \cdot h) = (f\#g) \cdot (f\#h) \tag{2} \]

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

The connection of \(\times\) and \# to prior discussions is simple. To allow for feature interactions, a sentence of a feature model (‘\(kj b\)’) is mapped to an expression by a \(\times\)-product of its terms (\(k \times j \times b\)). Equations (1) and (2) are used to reduce an expression with \(\times\) operations to an expression with only dot and \#, as below:

\[
\begin{align*}
p &= k \times j \times b & \text{// def of } p \\
&= k \times (j\#b \cdot j \cdot b) & \text{// (1)} \\
&= k\#(j\#b \cdot j \cdot b) \cdot k \cdot (j\#b \cdot j \cdot b) & \text{// (1)} \\
&= k\#j\#b \cdot k\#j \cdot k\#b \cdot k \cdot j\#b \cdot j \cdot b & \text{// (2)} \tag{4}
\end{align*}
\]

¹ Our Generic#Interface example is isomorphic to the classical example of fire and flood control [15]. 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 \(\cdot b\) and fire \(\cdot b\)) is straightforward. However, adding both (flood \(\cdot fire \cdot b\)) 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/extensions that are needed to make the flood and fire features work correctly together. The correct building design is flood\#fire\#flood\#fire \cdot b.

² A more general algebra has operations \(\times\), \#, and \()\) that are all commutative and associative [4]. This generality is not needed for this paper.

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 \(O(2^n)\) interactions (i.e. all possible feature combinations). Fortunately, the vast majority of feature interactions are empty, meaning that they correspond to the identity transformation 1, whose properties are:

\[ 1 \cdot f = f \cdot 1 = f \tag{3} \]

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

\[
\begin{align*}
\text{Generic} \times \text{Interface} \times \text{cFJ} &= \text{Generic#Interface#cFJ} \cdot \text{Generic#Interface} \\
&= \text{Generic#Interface#cFJ} \cdot \text{Generic#Interface#cFJ} \cdot \text{Interface#cFJ} \\
&= \text{Generic#Interface} \cdot \text{Generic#Interface#cFJ} \cdot \text{Interface#cFJ} \\
&= \text{Generic#Interface} \cdot \text{Generic#Interface} \cdot \text{cFJ}
\end{align*}
\]

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

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cFJ</td>
<td>core Featherweight Java</td>
</tr>
<tr>
<td>Cast</td>
<td>cast</td>
</tr>
<tr>
<td>Interface</td>
<td>interfaces</td>
</tr>
<tr>
<td>Generic</td>
<td>generics</td>
</tr>
<tr>
<td>Generic#Interface</td>
<td>generic and interface interactions</td>
</tr>
<tr>
<td>Generic#Cast</td>
<td>generic and cast interactions</td>
</tr>
</tbody>
</table>

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

4. Decomposing a Language into Features

We designed 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 [2].

All representations of a language (syntax, operational semantics, type system, proofs) are themselves 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 tech-
niques for extending language syntax are identical to ex-
tension techniques for the other representations. The critical
contribution of our approach is how we guarantee the cor-
rectness of composed proofs, the topic of Section 5.1.

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).

```
  e ::= x
  | e.f
  | e.m(ε)
  | new C(ε) ;

  (a)
```

```
  e ::= (C) e ;

  (b)
```

```
  e ::= x
  | e.f
  | e.m(ε)
  | new C(ε)
  | (C) e ;

  (c)
```

Figure 3: Union of Grammars

Modifying existing productions requires foresight to an-
ticipate 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 [1], 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 TPm added to method calls in
the cfJ expression grammar and TPt 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 TPm and TPt are empty, Figure 4c can be
inlined to produce the grammar in Figure 3c.

Now consider the changes that Generic makes to ex-
pression syntax: it redefines TPm and TPt 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.

Replacing an empty production with a non-empty one
is a standard programming practice in frameworks (e.g.
EJB [19]). 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.

```
  fields(C) = V \bar{T}
  \Gamma \vdash \pi : U \quad U : V
  \Gamma \vdash new C(\bar{e}) : C
  (T-NEW)
```

(a)

```
  fields(C) = V \bar{T}
  \Gamma \vdash \pi : U \quad U : V
  \Gamma \vdash new C(\bar{e}) : C
  (T-NEW)
  \Gamma \vdash e_0 : D \quad D : C
  \Gamma \vdash e_0 : C
  (T-UCAST)
```

(b)

Figure 4: Modification of Grammars

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.

```
fields(C) = V \bar{T}
\Gamma \vdash e_0 : D \quad D : C
\Gamma \vdash e_0 : C
(T-UCAST)
```

(c)
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 \(WF_c(D, TP_T C)\) predicate and for the \(D\) relational hole in the cFJ typing definitions. (Compare Figure 6a to its VP-extended counterpart 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 inline.

Different instantiations of VPs produce variations of the original productions and rules, with the lemma adapting accordingly. Figure 9 shows the VP instantiations and the corresponding statement for both cFJ and FGJ (\(e\) stands for empty in the cFJ case) with those instantiations inlined for clarity.

Without an accompanying proof, feature-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.

5. 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 10a-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 definitions from each of the features. For abstract predicates, the target predicate is the conjunction of all the VP definitions. The Coq encoding of expressions the Cast, and Generic features and the result of their composition with cFJ is given in Figure 11.

Figure 11: Coq Encoding of Fig. 3b and Fig. 5a-b.

In the discussion so far, composition has been strictly syntactic: definitions are directly unioned together or defaults are replaced. 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 [3], 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 case extension and VPs. 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.

Figure 12 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 cFJ and Cast. 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.

Figure 12: Syntax from cFJ and Cast Features and their Union.

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. Exp.WF from Fig. 10b ranges over the complete set of expressions from the final language, so its signature becomes $\forall e : Set, Context \rightarrow e' \rightarrow Ty \rightarrow Prop$. Of course, within a feature module these rules are written over the actual syntax definitions it provides. In order for the signatures to sync up, these
rules are parameterized by 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.

Parameterization also allows feature modules to include VPs, as shown in Figure 13. 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 parameter. In Figure 13, for example, FGJ.v builds the syntax for the target language by instantiating this VP with the definition of TP_m given in Generic.v. Alternatively, the syntax of cFJ can be built from the same inductive definition from cFJ using the default definition of TP_m it provides.

```
Definition TP_m := List Ty.
Inductive cFJ_e (e : Set) (TP_m : Set): Set :=
  e_var : Var => cFJ_e
  f_d_access : e => F => cFJ_e
  m_call : TP_m => e => M => List e => cFJ_e
  new : C => List e => cFJ_e.
```

Figure 13: Coq Syntax from cFJ with a Variation Point, and its Instantiation in FGJ.

### 5.1 Crafting Modular Proofs

Rather than writing multiple related proofs, our goal is to create a single proof for a generic statement of a theorem. This 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 5.1 (Figure 14). This lemma is then 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 14. The definition of these holes depends on the set of features included in the final language, so from the (human or computer) theorem prover’s point of view, they are opaque. Thus, this proof becomes stuck when it requires knowledge about behavior of \( \Phi_f \).

In order to proceed, the lemma must constrain possible VP instantiations to those that have the properties required by the proof. In the case of Lemma 5.1, this behavior is that \( \Phi_f \) must be the identity function for non-variable types and that \( \Phi_{SD} \) maps class types to class types. For this proof to hold for the target language, the instantiations of \( \Phi_f \) and \( \Phi_{SD} \) must have this property. More concretely, the proof assumes this behavior for all instantiations of \( \Phi_f \) and \( \Phi_{SD} \), producing the new generic Lemma 5.2. 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 it is written. The interface of a feature module is the union of all these assumptions together with the the set of lemmas about the behavior of its VP instantiations and definitions it provides. As long as the features included in the target language provide proofs satisfying this interface, a feature’s generic proofs can be specialized and reused in their entirety.

```
Lemma 5.1. If \( \Delta \vdash S \rightsquigarrow T \) and fields(\( \Phi_f(\Delta, T) \)) = \( \overline{\overline{T}} \),
then fields(\( \Phi_f(\Delta, S) \)) = \( \overline{\overline{S}} \) \( \forall_i S_i = T_i \) and \( g_i = f_i \) for all \( i \leq \#(f) \).
```

**Case S-Dir**

\( S = TP_0 C, CP_0 \) class \( C \) extends \( TP_1 D \{ S_0 \ldots \}, \)
\( T = \Phi_{SD}(TP_0, CP_0, TP_1 D) \).

By the rule F-CLASS, fields(\( \Phi_{SD}(TP_0, CP_0, TP_1 D) \)) = \( \overline{\overline{U}} \) with fields(\( TP_0 C \)) = \( \overline{\overline{U}} \). Assuming that for all class types \( TP_2 D' \), \( \Phi_f(\Delta, TP_2 D') \) = \( TP_2 D' \) and \( \Phi_{SD}(TP_0, CP_0, TP_2 D') \) returns a class type, \( \Phi_f(\Delta, \Phi_{SD}(TP_0, CP_0, TP_1 D)) = \Phi_{SD}(TP_0, CP_0, TP_1 D) \). It follows that \( \overline{\overline{T}} = \) fields(\( \Phi_{SD}(TP_0, CP_0, TP_1 D) \)) = \( \overline{\overline{U}} \) from which the conclusion is immediate.

Figure 14: Generic Statement of Lemmas 2.2 and 2.1 and Proof for S-Dir Case.

We also have to deal with new cases. Whenever a new rule or production is added, a new case must be added to proofs which induct over or case split 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 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.
5.2 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 variation 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 5.3 given in Figure 16. 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 Context Gamma Delta for any Context Delta. Any compatible extension of Context and app_context can thus reuse this proof.

**Lemma 5.3 (Subtype Weakening).** For all contexts \( \Gamma \) and \( \Delta \), if \( \Gamma \vdash S \subset T \), \( \Gamma \vdash S \subset \Gamma \).

Figure 15: Weakening lemma for subtyping.

To build the final proof, the target language inducts over subtype, as shown in the final box of Figure 16. For each constructor, the lemma dispatches to the proofs from the corresponding feature module. To reuse those proofs, each of their assumptions has to be fulfilled by a theorem (e.g., \texttt{TLookup.app} satisfies \texttt{TLookup.app}). The inductive hypothesis is provided to \texttt{cFJ.Weakensubtype.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 \texttt{cFJ subtype ind} induction scheme used to combine cFJ’s cases in the first box of Figure 16 is an example.

5.3 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 12. 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 12 and 13. Proofs for the target language are then built using the proofs from the constituent feature modules per the above discussion. Proof composition requires a straightforward 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.

6. Implementation of the FJ Product Line

We have 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 can built the seven variants on Featherweight Java listed in Section 3.2.3

<table>
<thead>
<tr>
<th>Module</th>
<th>Lines of Code in Coq</th>
</tr>
</thead>
<tbody>
<tr>
<td>cFJ</td>
<td>2612 LOC</td>
</tr>
<tr>
<td>Cast</td>
<td>463 LOC</td>
</tr>
<tr>
<td>Interface</td>
<td>499 LOC</td>
</tr>
<tr>
<td>Generic</td>
<td>6740 LOC</td>
</tr>
<tr>
<td>Generic#Interfaces</td>
<td>1632 LOC</td>
</tr>
<tr>
<td>Generic#Cast</td>
<td>296 LOC</td>
</tr>
</tbody>
</table>

**Figure 17: Feature Module Sizes**

While we achieve feature composition by manually instantiating these modules, the process is straightforward and should be mechanizable. Except for some trivial lemmas, the proofs for a 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. A 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 18 lists the certification times for feature modules and all the possible 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 18: Certification Times for Feature Modules and All Language Variants.**

7. Discussion and Future Work

Relying on parameterization for feature composition allows feature modules to be built and independently certified by Coq “out of the box” with the same level of assurance. With this approach, 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 [22], allows language designers to predict VPs in advance, thus enabling smooth and typically painless addition of new features.

What our work shows is a path for the structured evolution of languages. But of course, when unanticipated features are added using this style of composition, additional engineering may be required.

Existing definitions can be extended and reused as long as they already have the appropriate VPs and their inductive definitions are 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.

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 makes widespread changes throughout the base language (i.e. Generic), 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. $F \cdot G = G \cdot F$) their integration comes for free as their interaction module is empty (i.e. $F\#G = 1$).
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, replacing the original lemma and allowing the other proofs to be reused. In this manner, each proof is insulated from features which break the assumptions of other lemmas. 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.

Future work includes creating a new module-level composition operator that eases the burden of integrating new features. Ideally, this operator will allow subsequent features to extend a feature’s definitions with new cases or variations without modifying the feature and to provide patches to allow existing proofs to work with the extended definitions. As alluded to earlier, by operating at the module level, this operator would automate the tedious piece-by-piece composition currently employed to build target languages.

8. Related Work

The Tinkertype project [17] 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 [24]. 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 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 recomputed 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 recomputed. As long as the module’s assumptions hold, the proofs are guaranteed to hold for the final language.

Stärk et al. [27] develop a complete Java 1.0 compiler through incremental refinement of a set of Abstract State Machines. Starting with ExpL, 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 [2], noting that the proof could 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 [2] is completely by hand.

Thüm et al [29] 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 recomputed 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, as we have, enables a more semantic notion of composition.

The modular development of reduction rules are the focus of Mosses’ Modular Structural Operational Semantics [20]. 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:

\[
\frac{d \xrightarrow{X} d'}{\text{let } d \text{ in } e \xrightarrow{X} \text{let } d' \text{ in } e} \quad \text{(R-LETB)}
\]

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

\[
\frac{\{ p \leftarrow p_0 \mid p_1 \ldots \} \xrightarrow{e'} \{ p \leftarrow p_0 \mid p_1 \ldots \} \text{let } p_0 \text{ in } e}{\text{let } p_0 \text{ in } e \xrightarrow{e'} \text{let } p_0 \text{ in } e} \quad \text{(R-LETE)}
\]

These abstract labels correspond to the abstract contexts used by the cFJ subtyping rules to accommodate the updates of the Generic feature. 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 [9] and Mulhern [21] 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. 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 [10] proposes a using adaptive tactics written in Coq’s tactic definition language Ltac [11] 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 variation points.

9. 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 a product line. Decomposing proofs along feature boundaries enables the 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 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. Also we appreciate comments from Thomas Thüm and the referees on earlier drafts of this paper.

References


Fitting the Pieces Together:  
A Machine-Checked Model of Safe Composition

Benjamin Delaware, William R. Cook, Don Batory  
Department of Computer Sciences  
University of Texas at Austin  
Austin, Texas 78712 U.S.A.  
{bendy,wcook,batory}@cs.utexas.edu

ABSTRACT

Programs of a software product line can be synthesized by composing features which implement a unit of program functionality. In most product lines, only some combination of features are meaningful; feature models express the high-level domain constraints that govern feature compatibility. Product line developers also face the problem of safe composition — whether every product allowed by a feature model is type-safe when compiled and run. To study the problem of safe composition, we present Lightweight Feature Java (LFJ), an extension of Lightweight Java with support for features. We define a constraint-based type system for LFJ and prove its soundness using a full formalization of LFJ in Coq. In LFJ, soundness means that any composition of features that satisfies the typing constraints will generate a well-formed LJ program. If the constraints of a feature model imply these typing constraints then all programs allowed by the feature model are type-safe.

Categories and Subject Descriptors
F.3.3 [Studies of Program Constructs]: Type structure

General Terms  
Design, Languages

Keywords  
product lines, type safety, feature models

1. INTRODUCTION

Programs are typically developed over time by the accumulation of new features. However, many programs break away from this linear view of software development: removing a feature from a program when it is no longer useful, for example. It is also common to create and maintain multiple versions of a product with different sets of features. The result is a product line, a family of related products.

The inclusion, exclusion, and composition of features in a product line is easier if each feature is defined as a modular unit. A given feature may involve configuration settings, user interface changes, and control logic. As such, features typically cut across the normal class boundaries of programs. Modularizing a program into features, or feature modularity, is quite difficult as a result.

There are many systems for feature modularity based on Java, such as the AHEAD tool suite [5] and Classbox/J [7]. In these systems, a feature is a collection of Java class definitions and refinements. A class refinement is a modification to an existing class, adding new fields, new methods, and wrapping existing methods. When a feature is applied to a program, it introduces new classes to the program and its refinements are applied to the existing classes.

Figure 1 is a simple example of a product line containing two features, Bank and Sync. The Bank feature in Figure 1a implements an elementary Account class with a balance field and update method. Feature Sync in Figure 1b implements a synchronization feature so that accounts can be used in a multi-threaded environment. Sync has a refinement of class Account that modifies update to use a lock, which is intro-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 2009 ACM 978-1-60558-001-2/09/08 ...$5.00.

Figure 1: Account program: Sync•Bank

c) A composed program: Sync•Bank

feature Bank {  
class Account  
    extends Object{  
        int balance = 0;  
        void update(int x) {  
            int newBal = balance + x;  
            balance = newBal;  
        }  
    }  
}(a) Bank Feature

feature Sync {  
class Account  
    extends Object{  
        static Lock lock = new Lock();  
        refines void update(int x) {  
            lock.lock();  
            int newBal = lock.newLock();  
            lock.unlock();  
        }  
    }  
}(b) Synchronized Feature

class Account extends Object {  
    int balance = 0;  
    static Lock lock = new Lock();  
    void update(int x) {  
        lock.lock();  
        int newBal = lock.newLock();  
        lock.unlock();  
    }  
}(c) A composed program: Sync•Bank
duced as a static variable. Method refinement is accomplished by inheritance: Super.update() indicates a substitution of the prior definition of method update(x). Composing the refinement of Figure 1b with the class of Figure 1a produces a class that is equivalent to that in Figure 1c. The Bank feature can also be used on its own. While this example is simple, it exemplifies a feature-oriented approach to program synthesis: adding a feature means adding new members to existing classes and modifying existing methods. The following section presents a more complex example and more details on feature composition.

Not all features are compatible, and there may be complex dependencies among features. A feature model defines the legal combinations of features in a product line. A feature model can also represent user-level domain constraints that define which combinations of features are useful [9].

In addition to domain constraints, there are low-level implementation constraints that must also be satisfied. For example, a feature can reference a class, variable, or method that is defined in another feature. Safe composition guarantees that a program synthesized from a composition of features is type-safe. While it is possible to check individual programs by building and then compiling them, this is impractical. In a product line, there can be thousands of programs; it is more desirable to ensure that all legal programs are type-safe without enumerating the entire product line and compiling each program. This requires a novel approach to type checking.

We formalize feature-based product lines using an object-oriented kernel language extended with features, called Lightweight Feature Java (LFJ). LFJ is based on Lightweight Java [15], a subset of Java that includes a formalization in the Coq proof assistant [8], using the Ott tool [14]. A program in LFJ is a sequence 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 are separated by implicit interfaces that govern their composition. One solution to type checking these modules is to require explicit feature interfaces. We instead infer the necessary feature interfaces from the constraints generated by a constraint-based type system for LFJ. Regardless of whether we use feature interfaces or not, we would have to employ the same analysis to ensure safe composition. The type system and its safety are formalized in Coq. We then show how to relate the constraints produced by the type system to the constraints imposed by a feature model, using a reduction to propositional logic. This reduction allows us to statically verify that a feature model will only allow type-safe programs without having to generate and check each product individually.

2. SAFE COMPOSITION

Features can make significant changes to classes. Features can introduce new methods and fields to a class and alter the class hierarchy by changing the parent of a class. They can also refine existing methods by adding new statements before and after a method body or by replacing it altogether.

The features in Figure 2 illustrate how the Account class in the feature Bank can be modified. The RetirementAccount feature refines the Account class by updating its parent to Lehman, introducing a new field for a 401k account balance with an initial balance of 10000, and rewriting the definition for the update method to add x to the 401k balance. InvestmentAccount refines Account differently, updating its parent to WaMu, introducing a 401k field, and refining the update method to put half of x into a 401k before adding the rest to the original account balance.

A software product line can be modeled as an algebra that consists of a set of features and a composition operator •. We write M = {Bank, Investor, RetirementAccount, InvestmentAccount} to mean the product line M has the features declared above. One or more features of a product line build base programs through a set of class introductions:

Bank • a program with only the generic Account class
Investor • a program with only the AccountHolder class

The remaining features contain program refinements and extensions:

InvestmentAccount•Bank builds an investment account
RetirementAccount•Bank builds a retirement account where B•A is read as “feature B refines program A” or equivalently “feature B is added to program A”. A refinement can extend the program with new definitions or modify existing definitions. The design of a program is a composition of features called a product specification.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Class</th>
<th>Description</th>
<th>Figure</th>
</tr>
</thead>
<tbody>
<tr>
<td>RetirementAccount</td>
<td>Account extends Lehman</td>
<td>refines class Account extends Lehman, introducing a new field for a 401k account balance with an initial balance of 10000, and rewriting the definition for the update method to add x to the 401k balance.</td>
<td>Fig. 3</td>
</tr>
<tr>
<td>InvestmentAccount</td>
<td>Account extends WaMu</td>
<td>refines class Account extends WaMu, introducing a 401k field, and refining the update method to put half of x into a 401k before adding the rest to the original account balance.</td>
<td>Fig. 4</td>
</tr>
<tr>
<td>RetirementAccount•Bank</td>
<td>RetirementAccount•Bank</td>
<td>builds a retirement account where RetirementAccount refines Account differently, updating its parent to WaMu, introducing a 401k field, and refining the update method to put half of x into a 401k before adding the rest to the original account balance.</td>
<td>Fig. 5</td>
</tr>
</tbody>
</table>

This model of software product lines is based on stepwise development: one begins with a simple program (e.g., constant feature Bank) and builds more complex programs by progressively adding features (e.g., adding feature InvestmentAccount to Bank).

A set of n features can be composed in an exponential number of ways to build a set of order n! programs. A
class Account extends WaMu{
    int balance = 0;
    int 401kbalance = 0;
    void update(int x) {
        x = x/2;
        int newBal = balance + x;
        balance = newBal;
        401kbalance += x;
    }
}

Figure 4: InvestmentAccount•Bank

class Account extends Lehman{
    int balance = 0;
    int 401kbalance = 10000;
    void update(int x) {
        401kbalance += x;
    }
}

class AccountHolder extends Object {
    Account a = new Account();
    void payday (int x; int bonus) {
        a.401kbalance += bonus;
        return a.update(x);
    }
}

Figure 5: RetirementAccount•Investor•Bank

composition might fail to meet the dependencies of its constituent features, so only a subset of the programs built from this set of features is well-typed. The feature model defines the set of programs which belong to a product line by constraining the ways in which features can be composed. The goal of safe composition is to ensure that the product line described by a feature model is contained in the set of well-typed programs, i.e., that all of its programs are well-typed.

The combinatorial nature of product lines presents a number of problems to determining safe composition. The members and methods of a class referenced in a feature might be introduced in several different features. Consider the AccountHolder class introduced in the Investor feature: this account holder is the employee of a company which gives a small bonus with each paycheck which the employee adds directly into the 401k balance in his account. In order for a composition including the Investor feature to build a well-typed Java program, it must be composed with a feature that introduces this field to the Account class, in this case either InvestmentAccount or RetirementAccount. This requirement could also be met by a feature which sets the parent of Account to a different class from which it inherits the 401kbalance field. Since a parent of a class can change through refinement, the inherited fields and methods of the classes in a feature are dependent on a specific product specification. Each feature has a set of type-safety constraints which can be met by the combination of a number of different features, each with their own set of constraints. To study the interaction of feature composition and type safety, we first develop a model of Java with features.

3. 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 [15]. 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. 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 [12] 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 have extended to prove the soundness of LFJ mechanically. The proof scripts for the system are available at http://www.cs.utexas.edu/~bendy/featurejava.php.

The syntax extensions LFJ adds to LJ in order to support feature-oriented programming are given in Figure 6. The syntax of LFJ is modelled after the feature-oriented extensions to Java used in the AHEAD tool suite. A feature definition $FD$ maps a feature name $F$ to a list of class declarations $FD_cld$ and a list of class refinements $rmd$. A class refinement $rcld$ includes a class name $C$, a set of LJ field and method introductions, $\overline{FD_cld}$ and $\overline{rmd}$, and the name of the updated parent class $cl$. A method refinement advises a method with signature $ms$ with two lists of LJ statements $\overline{t}$ 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 used by a product line. A product specification $PS$ selects a distinct list of feature names from the feature table.

3.1 Feature Composition

In LJ, a program $P$ is a set of class definitions. The $\bullet$ operator composes a feature $FD = \text{feature } F \{cld; rcld\}$ with an LJ program $P$ to build a refined program:

$$FD \bullet P = \{cld\} \cup \{rcld \bullet cld \mid cld \in P \land \text{id}(cld) \notin \text{id}s(rcld)\}$$ (1)

Composition builds a refined program by first introducing the class definitions in $cld$, replacing any classes in $P$ which share an identifier with a class in $cld$. The remaining classes in $P$ are added to this set after applying the refinements in $rcld$ using the $\bullet$ operator. For all classes $cld \in P$ with
an identifier not refined by \( \overline{rcld} \), is simply the identity function. If a class refinement \( \overline{rcld} \) in \( rcld \) has the same identifier as \( cld \), \( \cdot \) builds the 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 \( rcld \). Composition fails if \( P \) lacks a class refined by \( rcld \) or if a class refined by \( rcld \) lacks a method which is 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 LJF 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 define the same class with a different parent.

4. Typechecking Feature Models

A feature model is safe if it only allows the creation of well-formed LJ programs. Any particular product specification can be checked by composing its features and then checking the type safety of the resulting program in the standard LJ type system. A naive approach to checking the safety of a feature model is simply to iterate over all the programs it describes, type checking each individually. This approach constructs a potentially exponential number of programs, making it computationally expensive. Instead, we propose a type system which allows us to statically verify that all programs described by a feature model are type-safe

The key difficulty with this approach is that features are typically program fragments which make use of class definitions made in other features; these external dependencies can only be resolved during composition with other features. Every LJ construct has two categories of requirements which must be met in order for it to be well-formed in the LJ type system. The first category consists of premises which only depend on the structure of the construct, e.g. the requirement that the parameters of a well-formed method be distinct. The remaining premises access information from the surrounding program to the path \( P \times C \rightarrow cld \) function which maps identifiers to their definitions in \( P \). For example, when assigning \( y \) to \( x \) in a method body, the path function is used to determine that the type of \( y \) is a subtype of the type of variable \( x \). Intuitively, these premises define the structure of the programs in which LJ constructs are well-formed. In the standard LJ type system, the structure of the surrounding program is known. In a software product line, however, each feature can be included in a number of programs, and the final makeup of the surrounding program depends on the other features in a product specification. Converting these kinds of premises into constraints provides an explicit interface for an LJ construct with any surrounding program. A feature’s interface determines which features must be included in a product specification in order for its constructs to be well-formed in the final LJ program.

4.1 LFJ Type System

In this section, we present a constraint-based type system for LFJ. In order to relate this to the LJ type system, we have also developed a constraint-based type system for LJ. Both these systems 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 consideration of which product specifications have a combination of features satisfying them.

The constraints used to type LJ and LFJ, listed in Figure 7, 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**

\[
C \text{ introduces } m.s \text{ before } F \\
C \text{ introduced before } F
\]

**Uniqueness Constraints**

\[
cld f \text{ unique in } C \\
cld m (\text{val}_k) \text{ unique in } C
\]

**Structural Constraints**

\[
cld_1 < cld_2 \\
dcl_2 < ftype(cld_1, f) \\
ftype(cld_1, f) < dcl_2 \\
mtype(cld, m) < cld_k \rightarrow cld \\
defined(cld) \\
f \notin \text{fields(parent}(C))
\]

**Feature Constraint**

\[
\text{In}_{F} \Rightarrow \xi_k
\]

Figure 7: Syntax of Lightweight Feature Java typing constraints.

The typing rules for LFJ are found in Figure 8-10 and rely on judgements of the form \( \vdash J \mid \xi \), where \( J \) is a typing judgement from LFJ, and \( \xi \) is a set of constraints. \( \xi \) provides an explicit interface which guarantees that \( J \) holds in any product specification that satisfies \( \xi \). Typing judgements for statements include a context \( \Gamma \) mapping variable names to their types. 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 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.

\[ \Gamma \vdash s \mid C \] Statement well-formed in context subject to constraints

\[ \begin{align*}
\Gamma &\vdash s_k \mid C_k \\
\Gamma &\vdash \{s_k\} \mid \bigcup_k C_k
\end{align*} \] (WF-BLOCK)

\[ \Gamma(x) = \tau_1 \quad \Gamma(var) = \tau_2 \\
\Gamma \vdash \text{var} = x; \{\tau_1 < \tau_2\} \] (WF-VAR-ASSIGN)

\[ \begin{align*}
\Gamma(x) = \tau_1 \\
\Gamma(var) = \tau_2 \\
\Gamma \vdash \text{var} = x.f; \{\text{ftype}\}(\tau_1, f) \times \tau_2\} \] (WF-FIELD-READ)

\[ \begin{align*}
\Gamma(x) = \tau_1 \\
\Gamma(y) = \tau_2 \\
\Gamma \vdash \text{var} = x.f; \{\text{ftype}\}(\tau_1, f)\} \] (WF-FIELD-WRITE)

\[ \begin{align*}
\Gamma(x) = \tau_1 \\
\Gamma(y) = \tau_2 \\
\Gamma \vdash s_1 \mid C_1 \\
\Gamma \vdash s_2 \mid C_2 \\
C_3 = \{\tau_2 < \tau_1 \vee \tau_1 < \tau_2\} \\
\Gamma \vdash \text{if } x== \text{y then } s_1 \text{ else } s_2 \mid C_1 \cup C_2 \cup C_3 \] (WF-IF)

\[ \begin{align*}
\Gamma(var) = \tau_1 \\
\Gamma \vdash \text{var} = \text{new cl}() \mid \{\tau_2 < \tau_1\} \] (WF-NEW)

\[ \Gamma(y_k) = \pi_k \\
C = \{\text{mtype}\}(\tau, \text{meth}) \rightarrow \pi \} \\
\Gamma \vdash \text{var} = x.meth(y_k) \mid C \] (WF-MCALL)

Figure 8: Typing Rules for LJ and LFJ statements.

Once the constraints \( C \) for a feature table are generated according to the rules in Figure 10, we can check whether a specific product specification \( PS \) satisfies \( C \) using the rules in Figure 11. Satisfaction of the structural constraints is given for LJ and uses the \textit{path} function. Satisfaction of the structural constraints in LFJ replaces \textit{path} with the \textit{CT} function which mimics the behavior of \textit{path} on a composed product specification without having to build the product:

\[ CT((F, PS), C) = \begin{cases} 
\text{path}(\text{cl}d, C) & \text{C} \in \text{ids}(\text{cl}d) \\
\text{rcl}d \cdot CT(PS, C) & \text{C} \notin \text{ids}(\text{cl}d)
\end{cases} \] (2)

where \textit{feature} \( F \{\text{cl}d, \text{rcl}d\} \in FT \). Compositional constraints on a feature \( F \) are satisfied when a feature which introduces the named class or method 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 \( C \). Feature constraints on a \( F \) are satisfied when \( F \) is not included in \( PS \) or \( PS \models \xi_k \).

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 can rely on definitions which are introduced in a later feature or have classes used to satisfy typing constraints which could also be overriden by a subsequent feature. For this reason, our type system only considers final product specifications, making no guarantees about the behavior of intermediate programs.

### 4.2 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 system given in [15]. 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 specifications 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.

\[ \vdash P \mid C \] Program well-formed subject to constraints

\[ \vdash \text{cl}d_k \mid C_k \quad P = \text{cl}d_k \] (distinct ids (P))

\[ \vdash P \mid \bigcup_k C_k \] (WF-PROGRAM)

\[ \vdash \text{FD} \mid C \] Feature well-formed subject to constraints

\[ \vdash \text{cl}d_k \mid C_k \quad \vdash \text{rcl}d_k \mid C_k \] (WF-FEATURE)

\[ \vdash \text{FD}_k \mid C_k \] (WF-FEATURE-TABLE)

Figure 10: Typing Rules for LFJ Programs and Features.
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 4.2 (Soundness of LFJ Type System).** Let $PS$ be an LFJ product specification for feature table $FT$ and $C$ be a set of constraints such that $\vdash FT \mid C$. If $PS \models C$, then the 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 11, 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 $PS$.

(i) Let $PS$ be a product specification for feature table $FT$ and $C$ be a set of constraints such that $\vdash FT \mid C$. If $PS \models C$, composition of the features in $PS$ produces a valid $LJ$ program, $P$.

(ii) Given (i), $P$ is typeable in the constraint-based $LJ$ type system with constraints $C'$.

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. 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 constraint used in typing both kinds of statements in the same set as that used for the method in the final composition. This does not entail that \( C = C' \), however, as there could be some construct in \( PS \) that is overwritten by an introduction in a subsequent feature.

(b) For any structural constraint \( K \), if \( PS \models K \), then \( P \models K \).

This reduces to showing that class declaration returned by \( CT(PS, C) \) is the same that returned by \( path(P, C) \). This follows from tracing the definition of the \( CT \) function down to the final introduction of \( C \) 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 helper functions which call \( path \) in \( P \) (i.e. \( ftype, \text{mtype} \)) and those that use \( CT \) in \( PS \) return the same values. Thus, the satisfaction judgements for \( PS \) and \( P \) are equivalent.

All constraints in \( C' \) appear in \( C \), so \( PS \models C' \). By (b) above, it follows that \( P \models C' \). A \( type(C) < type(Object) \) is generated for each class in \( P \), so \( P \) has a well-founded, acyclic class hierarchy. Furthermore, the definition of composition ensures that all classes in \( P \) have distinct names. By Theorem 4.1, \( P \) must be a well-formed LJ program.

5. TYPE-CHECKING PRODUCT LINES

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. These constraints remain static regardless of the product specification being checked, as does the feature model used to describe members of a product line. We now show how to relate the product specifications described by the two by expressing both as propositional formulas. Checking safe composition of a product line amounts to showing that the programs allowed by the feature model are contained within the set of type-safe products.

Feature models are compact representations of propositional formulas [6], making propositional logic a natural formalism in which to relate feature models to the constraints generated by the LFJ type system. The variables used in the propositional representation of feature models have the form of the first two entries in Figure 12. A product line is described by the satisfying assignments to its feature model. The designer of the product line from the introduction might want it to only include a specialized account class, which they could express with the feature model: \( \text{InAccount} \rightarrow (\text{InvestmentAccount} \lor \text{RetirementAccount}) \).

\( \text{In} \) : Feature \( A \) is included.
\( \text{Prec}_{A,B} \) : Feature \( A \) precedes Feature \( B \).
\( \text{Sty}_{\tau_1, \tau_2} \) : \( \tau_1 \) is a subtype of \( \tau_2 \).

Figure 12: Description of propositional variables.
5.1 Safety of Feature Models

We now consider how to use the constraints generated by the LFJ type system to describe type-safe product specifications in propositional logic. A product specification satisfies an included feature's constraints by satisfying each of the constraints on its constructs. We can leverage this by translating each constraint into a propositional description of the product specifications which satisfy it. The set of product specifications which satisfy the constraints on a feature \( F \) is one that satisfies the conjunction of those formulas, \( C_F \). Thus, the set of well-typed product specifications is described by \( \bigwedge F \text{ In}_F \rightarrow C_F \).

The rules for translating constraints are given in Figure 13. The translation of the compositional, uniqueness and feature constraints is straightforward. Structural constraints enforce two important properties: inclusion of classes and class members and subtyping. A product specification satisfies the former if some included feature introduces that construct and it is not overwritten by the reintroduction of a class. This is enforced by the \( \text{Final}_L \) predicate which holds when \( F \) is not followed by a feature \( G \) which reintroduces \( cl \) and the \( \text{FinalIn}_L \) predicate which further requires that \( F \) does not overwrite \( cl \) if it redefines it. Subtyping constraints are represented by the \( \text{Sty}_{r,v} \) variables. Truth assignments to these variables are forced to respect the class hierarchy of a product specification by the final four rules in Figure 14. In effect, the \( \text{Sty}_{\text{Total}} \) rule builds the transitive closure of the subtyping relation, starting with the parent/child relationships established by the last definition of a class in a product specification.

Our formulas include the \( \text{Prec} \) variables to capture feature ordering in product specifications because it affects composition. A truth assignment must respect the properties of the precedence relation in order for it to represent valid product specifications. The first four formulas in Figure 14 impose these properties: the precedence relation must be transitive, irreflexive, antisymmetric, and total on all features included in a product specification. A satisfying assignment to the conjunction of all the constraints in Figure 14, \( \text{WF}_{\text{Spec}} \), obeys the properties of the precedence and subtyping relations, and thus corresponds to a unique product specification.

In order to type-check a product line, we first generate the constraints on a feature table using the LFJ typesystem. We then translate these constraints according to the rules in Figure 13, building a formula \( \xi \) describing the set of type-safe programs. With this in hand, checking that a product line is contained in the set of type-safe programs is reduced to checking the validity of \( \text{WF}_{\text{Spec}} \wedge FM \rightarrow \xi \). The left side of the implication restricts truth assignments to valid product specifications which are allowed by the feature model \( FM \), while the right side ensures that a 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 a typing problem.

5.2 Feasibility of Our Approach

While checking the validity of \( FM \wedge \text{WF}_{\text{Spec}} \rightarrow \phi_{\text{safe}} \) is co-NP-complete, the SAT instances generated by our approach are highly structured, making them amenable to fast analysis by modern SAT solvers. We have previously implemented a system based on our approach for checking safe composition of AHEAD software product lines [16]. The size statistics for the four product lines analyzed are presented in Table 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. The term Jak in Table 1 refers to the Jakarta language (the basis for LFJ).

<table>
<thead>
<tr>
<th>Product Line</th>
<th># of Features</th>
<th># of Prog.</th>
<th>Code Base</th>
<th>Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>BPL</td>
<td>7</td>
<td>20</td>
<td>2000/2000</td>
<td>1K/1K</td>
</tr>
<tr>
<td>GPL</td>
<td>17</td>
<td>8</td>
<td>12K/16K</td>
<td>8K/12K</td>
</tr>
<tr>
<td>GPL</td>
<td>18</td>
<td>80</td>
<td>1800/1800</td>
<td>700/700</td>
</tr>
<tr>
<td>JPL</td>
<td>70</td>
<td>56</td>
<td>34K/48K</td>
<td>22K/35K</td>
</tr>
</tbody>
</table>

Table 1: Product Line Statistics from [12].

6. RELATED WORK

An earlier draft of this paper was presented at FOAL [11]. Our strategy of representing feature models as propositional formulas in order to verify their consistency was first proposed in [6]. 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 [10] 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 [6], 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.

The previous two approaches relied on user-provided constraints when validating feature models. The genesis of our current approach was a system developed by Thaker et al. [16] 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.

If features are thought of as modules, the feature model used to describe a product line is a module interconnection language [13]. Normally, the typing requirements for a mod-
where constraints during a linking phase which combines individu
over type variables, similar to the constraints used in the
fragments augmented with typing constraints that ranged
gram. These code fragments were compiled into bytecode
Like features, the source code fragments they considered
to type check, compile, and link source code fragments [1].

erence with normal linking is that we check all combina-
products specifica. If composition in a linking process
interface” for each module. We instead infer a module’s “re-
defined

\[ \tau_1 \prec \tau_2 \]

\[ \tau_2 \prec \text{ftype}(\tau_1, f) \]

\[ \forall \{ \text{Sty}_{\tau_2,e} \land \text{Sty}_{\tau_1,\text{type}(\text{cl})} \land \text{FinalIn}_{\text{id}(\text{cl}), F} \mid \exists \text{cl} \in \text{clds}(F) \land \text{cl} \neq \text{cl} \} \land \]
in a product line. This feature model is safe if it only allows the construction of well-formed programs. Simply enumerating all the programs (feature combinations) described by the feature model is computationally expensive and impractical for large product lines. In order to statically verify that a product line is safe, we have developed a calculus for studying feature composition in Java and a constraint-based type system for this language. The constraints generated by the typing rules provide an interface for each feature. We have shown that the set of constraints generated by our type system is sound with respect to LJ’s type system. We verify the correctness of the type checking of the product line using propositional formulas. In Software Product Lines Conference, LNCS 3714, pages 7–20. Springer, 2005.


