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.

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

(a) Bank Feature

```
feature Sync {
  refines class Account
      extends Object{
    static Lock lock
          = new Lock();
    refines void update(int x) {
      lock.lock();
      Super.update();
      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 = balance + x;
    balance = newBal;
    lock.unlock(); }}}
```

(c) A composed program: Sync•Bank

Figure 1: Account with synchronization feature

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-

[*] This material is based upon work supported by the National Science Foundation under Grant CCF-0724979.

ESEC-FSE'09, August 23–28, 2009, Amsterdam, The Netherlands.
Copyright 2009 ACM 978-1-60558-001-2/09/08 ...$5.00.

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 a 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.

```
feature InvestmentAccount {
  refines class Account extends WaMu {
    int 401kbalance = 0;
    refines void update (int x) {
      x = x/2; Super.update(); 401kbalance += x;
  }}}

feature RetirementAccount {
  refines class Account extends Lehman {
    int 401kbalance = 10000;
    int update (int x) {
      401kbalance += x;
  }}}

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

Figure 2: Definitions of InvestmentAccount, Investor, and RetirementAccount features.

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

Figure 3: RetirementAccount•Bank

A software product line can be modelled 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 $B \bullet A$ 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*.

P_1 = RetirementAccount•Bank Fig. 3
P_2 = InvestmentAccount•Bank Fig. 4
P_3 = RetirementAccount•Investor•Bank Fig. 5

This model of software product lines is based on step-wise 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`.

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

Figure 6: Modified Syntax of Lightweight Feature Java.

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 $\overline{cld}$ and a list of class refinements $\overline{rcld}$. A class refinement $rcld$ includes a class name C, a set of LJ field and method introductions, $\overline{fd}$ and $\overline{md}$, a set of method refinements $\overline{rmd}$, and the name of the updated parent class cl. A method refinement advises a method with signature ms with two lists of LJ statements $\overline{s}$ and an updated return value y. When applied to an existing method, a method refinement wraps the existing method body with the advice. The parameters of the original method are passed implicitly because the refinement has the same signature as the method it refines. The feature table FT contains the set of features 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 $FD = \textbf{feature } F \; \{\overline{cld}; \; \overline{rcld}\}$ with an LJ program P to build a refined program:

$$FD \bullet P = \{\overline{cld}\} \cup \{\overline{rcld} \cdot cld \mid cld \in P \wedge \mathbf{id}(cld) \notin \mathbf{ids}(\overline{cld})\} \quad (1)$$

Composition builds a refined program by first introducing the class definitions in $\overline{cld}$, replacing any classes in P which share an identifier with a class in $\overline{cld}$. The remaining classes in P are added to this set after applying the refinements in $\overline{rcld}$ using the $\cdot$ operator. For all classes $cld \in P$ with an identifier not refined by $\overline{rcld}$, $\cdot$ is simply the identity function. If a class refinement $rcld$ in $\overline{rcld}$ 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 $\overline{rcld}$ 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 LFJ feature table can construct a family of programs through composition, with the set of class definitions determined by the sequence of features which produced them. The class hierarchy is also potentially different in each program: refinements can alter the parent of a class, and two mutually exclusive features can 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 without having to synthesize the entire family of programs.

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 through 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 **introduces** ms **before** F
C **introduced before** F

Uniqueness Constraints

$cl\ f$ **unique in** C
$cl\ m\ (\overline{vd_k}^k)$ **unique in** C

Structural Constraints

$cl_1 \prec cl_2$
$cl_2 \prec \mathbf{ftype}(cl_1, f)$
$\mathbf{ftype}(cl_1, f) \prec cl_2$
$\mathbf{mtype}(cl, m) \prec \overline{cl_k}^k \rightarrow cl$
$\mathbf{defined}(cl)$
$f \notin \mathbf{fields}(\mathbf{parent}(C))$
$\mathbf{pmtype}(C, m) = \tau$

Feature Constraint

$\mathbf{In}_F \Rightarrow \overline{\xi_k}^k$

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 ξ is a set of constraints. ξ provides an explicit interface which guarantees that J holds in any product specification that satisfies ξ. Typing judgements for statements include a context Γ 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.

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

$$\frac{\overline{\Gamma \vdash s_k \mid \mathcal{C}_k}^{k}}{\Gamma \vdash \{s_k\} \mid \bigcup_k \mathcal{C}_k} \qquad \text{(WF-}\textsc{Block}\text{)}$$

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

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

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

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

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

$$\frac{\begin{array}{c}\Gamma(x) = \tau \qquad \Gamma(var) = \pi \qquad \overline{\Gamma(y_k) = \pi_k}^{k} \\ \mathcal{C} = \{\mathbf{mtype}(\tau, meth) \prec \overline{\pi_k}^{k} \rightarrow \pi\}\end{array}}{\Gamma \vdash var = x.meth(\overline{y_k}^{k}) \ \mid \mathcal{C}} \qquad \text{(WF-MC}\textsc{all}\text{)}$$

Figure 8: Typing Rules for LJ and LFJ statements.

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

$$CT((F, PS), C) = \begin{cases} \mathbf{path}(\overline{cld}, C) & C \in \mathbf{ids}(\overline{cld}) \\ \overline{rcld} \cdot CT(PS, C) & C \notin \mathbf{ids}(\overline{cld}) \end{cases} \qquad (2)$$

where **feature** F $\{\overline{cld}, \overline{rcld}\} \in FT$. 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 $\overline{PS \models \xi_k}^{k}$.

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

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

$$\frac{\begin{array}{c}\overline{\vdash cld_k \mid \mathcal{C}_k}^{k} \qquad P = \overline{cld_k}^{k} \\ \mathbf{distinct\ ids}\ (P)\end{array}}{\vdash P \mid \bigcup_k \mathcal{C}_k} \qquad \text{(WF-}\textsc{Program}\text{)}$$

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

$$\frac{\overline{\vdash cld_k \mid \mathcal{C}_k}^{k} \qquad \overline{\vdash_F rcld_\ell \mid \mathcal{C}_\ell}^{\ell}}{\vdash \mathbf{feature}\ F\ \{\overline{cld_k}^{k}\overline{rcld_\ell}^{\ell}\} \mid \mathbf{In}_F \Rightarrow \bigcup_k \mathcal{C}_k \cup \bigcup_\ell \mathcal{C}_\ell} \qquad \text{(WF-}\textsc{Feature}\text{)}$$

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

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

Figure 10: Typing Rules for LFJ Programs and Features.

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.

Theorem 4.1 (Soundness of constraint-based LJ Type System)**.** *Let P be an LJ program with distinct class names and an acyclic, well-founded class hierarchy. Let $\mathcal{C}$ be the set of constraints generated by a class cld in P: $\vdash cld \mid \mathcal{C}$. cld is well-formed if and only if P satisfies $\mathcal{C}$: $P \vdash cld \leftrightarrow P \models \mathcal{C}$.*

Proof. The two key pieces of this proof are: showing that satisfaction of each of the constraints guarantees that the corresponding judgement holds, and that there is a one-to-one correspondence between the constraints generated by the typing rules in Figure 9 and the external premises used in the declarative LJ type system. The former is straightforward except for the subtyping constraint, which relies on the **path** function to check for satisfaction. We can prove their equivalence by induction on the derivation of the subtyping

$\boxed{\vdash_{\tau,F}\ md \mid \mathcal{C}}$ Method well-formed in class subject to constraints

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

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

$$\frac{\begin{array}{c}\mathbf{distinct}(\overline{f_j}) \qquad \mathbf{distinct}(\overline{m_k}) \qquad C \neq cl \qquad \mathbf{type}(C) = \tau \qquad \overline{\vdash_\tau cl_k\ meth_k\ (\overline{cl_{\ell,k}\ var_{\ell,k}}^{\ell})\ mb_k \mid \mathcal{C}_k}^{k} \\ \xi = \bigcup_j \{f_j \notin \mathbf{fields}(\mathbf{parent}(C))\} \qquad \upsilon = \bigcup_j \{cl_j\ f_j\ \mathbf{unique\ in}\ C\} \qquad \upsilon' = \bigcup_k \{cl_k\ meth_k\ (\overline{cl_{\ell,k}\ var_{\ell,k}}^{\ell})\ \mathbf{unique\ in}\ C\} \\ \tau \prec \mathbf{type}(\mathbf{Object}) \qquad \xi' = \bigcup_k \{\mathbf{pmtype}(C, meth_k) = \overline{cl_{\ell,k}}^{\ell} \rightarrow cl_k\}\end{array}}{\vdash \mathbf{class}\ C\ \mathbf{extends}\ cl\ \{\overline{cl_j\ f_j}^{j};\ \overline{cl_k\ meth_k\ (\overline{cl_{\ell,k}\ var_{\ell,k}}^{\ell,k})\ mb_k}^{k}\} \mid \bigcup_k \mathcal{C}_k \cup \{\mathbf{defined}\ cl, \overline{\mathbf{defined}\ cl_j}^{j}\} \cup \xi \cup \xi' \cup \upsilon \cup \upsilon'}$$

(WF-Class)

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

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

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

$$\frac{\begin{array}{c}C \neq cl \qquad \mathbf{type}(C) = \tau \qquad \overline{\vdash_\tau cl_k\ meth_k\ (\overline{cl_{\ell,k}\ var_{\ell,k}}^{\ell})\ mb_k \mid \mathcal{C}_k}^{k} \qquad \overline{\vdash_{\tau,F} rmd_m \mid \mathcal{C}'_m}^{m} \\ \xi = \bigcup_j \{f_j \notin \mathbf{fields}(\mathbf{parent}(C))\} \qquad \upsilon = \bigcup_j \{cl_j\ f_j\ \mathbf{unique\ in}\ C\} \qquad \upsilon' = \bigcup_k \{cl_k\ meth_k\ (\overline{cl_{\ell,k}\ var_{\ell,k}}^{\ell})\ \mathbf{unique\ in}\ C\} \\ \xi' = \bigcup_k \{\mathbf{pmtype}(C, meth_k) = \overline{cl_{\ell,k}}^{\ell} \rightarrow cl_k\}\end{array}}{\begin{array}{c}\vdash_F \mathbf{refines\ class}\ C\ \mathbf{extending}\ cl\ \{\overline{cl_j\ f_j}^{j}; \overline{\vdash_\tau cl_k\ meth_k\ (\overline{cl_{\ell,k}\ var_{\ell,k}}^{\ell,k})\ mb_k}^{k};\ \overline{rmd_{\ell,k}}^{\ell,k}\} \mid \bigcup_k \mathcal{C}_k \cup \bigcup_m \mathcal{C}'_m \cup \\ \{\mathbf{defined}\ cl, \overline{\mathbf{defined}\ cl_j}^{j}, C\ \mathbf{introduced\ before}\ F,\} \cup \xi \cup \xi' \cup \upsilon \cup \upsilon'\end{array}}$$

(WF-Refines-Class)

Figure 9: Typing Rules for LFJ method and class refinements.

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 $\mathcal{C}$ be a set of constraints such that $\vdash FT \mid \mathcal{C}$. If $PS \models \mathcal{C}$, then the composition of the features in PS produces a valid, well-formed LJ program.*

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

(i) Let PS be a product specification for feature table FT and $\mathcal{C}$ be a set of constraints such that $\vdash FT \mid \mathcal{C}$. If $PS \models \mathcal{C}$, 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.

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

In essence, we must show that the premises of the constraint-based LJ typing judgements hold. Our assumption that each class in PS is a descendant of **Object** ensures that P has an acyclic, well-founded class hierarchy. The premises for

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

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

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

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

$$\frac{\tau_2\in\mathbf{path}(P,\tau_1)}{P\models\tau_1\prec\tau_2} \qquad \frac{\mathbf{ftype}(P,\mathbf{parent}(C),f)=\bot}{P\models f\notin\mathbf{fields}(\mathbf{parent}(C))}$$

$$\frac{\begin{array}{c}\mathbf{mtype}(P,\mathbf{parent}(C),m)=\bot\ \vee\\ \mathbf{mtype}(P,\mathbf{parent}(C),m)=\tau\end{array}}{P\models\mathbf{pmtype}(C,m)=\tau}$$

$$\frac{\begin{array}{c}\tau.ms\in H \qquad \tau\notin\mathbf{introductions}(\overline{B_\ell}^{\ell})\\ PS=\overline{A_k}^{k}F\overline{B_\ell}^{\ell}H\overline{C_j}^{j}\end{array}}{PS\models\tau\ \mathbf{introduces}\ ms\ \mathbf{before}\ F}$$

$$\frac{PS=\overline{A_k}^{k}F\overline{B_\ell}^{\ell}H\overline{C_j}^{j} \qquad \mathbf{C}\in H}{PS\models C\ \mathbf{introduced\ before}\ F}$$

$$\frac{\begin{array}{c}\mathbf{type}(C)=\tau\\ \forall A,B\in PS,\tau.cl_1\ f\in A\wedge\tau.cl_2\ f\in B\to cl_1=cl_2\end{array}}{PS\models cl\ f\ \mathbf{unique\ in}\ C}$$

$$\frac{\begin{array}{c}\mathbf{type}(C)=\tau \qquad ms_1=cl\ m\ (\overline{vd_k}^{k})\\ ms_2=cl'\ m\ (\overline{vd'_k}^{k})\\ \forall A,B\in PS,\tau.ms_1\in A\tau.ms_2\in B\to ms_1=ms_2\end{array}}{PS\models cl\ m\ (\overline{vd_k}^{k})\ \mathbf{unique\ in}\ C}$$

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

Figure 11: Satisfaction of typing constraints.

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 constraints, which are satisfied when a method or field name is only introduced by a single feature. Since $PS \models \mathcal{C}$, it follows that the premises of the LJ typing rules hold for P and that there exists a set of constraints $\mathcal{C}'$ such that $\vdash P \mid \mathcal{C}'$.

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

We break this proof into two pieces:

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

The key observation for this proof is that every class, method, and statement in P originated from some feature in PS. Thus, for any constraint on a construct in P, there is a corresponding constraint on the feature in PS that generated that construct. The most interesting case is for the constraints generated by method bodies: a statement contained in a method body can come from either the initial introduction of that method or advice added by a method refinement. In either case, the statement was included in some feature in PS and thus generated some set of constraints in $\mathcal{C}$. Because method signatures are fixed across refinement, the context used in typing both kinds of statements is the same as that used for the method in the final composition. This does not entail that $\mathcal{C} = \mathcal{C}'$, however, as there could be some construct in PS that is overwritten by an introduction in a subsequent feature.

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

This reduces to showing that class declaration returned by $CT(PS, C)$ is the same that returned by $\mathbf{path}(P, C)$. 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 $\cdot$ operator used recursively to build CT; each refinement succeeds by (i) above. Since the two functions are the same, the helper functions which call **path** in P (i.e. **ftype**, **mtype**) and those that use CT in PS return the same values. Thus, the satisfaction judgements for PS and P are equivalent.

All constraints in $\mathcal{C}'$ appear in $\mathcal{C}$, so $PS \models \mathcal{C}'$. By (b) above, it follows that $P \models \mathcal{C}'$. A $\mathbf{type}(C) \prec \mathbf{type}(\mathbf{Object})$ is generated for each class in P, so P has a well-founded, acyclic class hierarchy. Futhermore, 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: $\mathbf{In}_{\mathsf{Account}} \to (\mathbf{In}_{\mathsf{InvestmentAccount}} \vee \mathbf{In}_{\mathsf{RetirementAccount}})$.

$\mathbf{In}_A$	: Feature A is included.
$\mathbf{Prec}_{A,B}$	: Feature A precedes Feature B.
$\mathbf{Sty}_{\tau_1,\tau_2}$	: τ_1 is a subtype of τ_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 satsifies the conjuction of those formulas, $\mathcal{C}_F$. Thus, the set of well-typed product specifications is described by $\bigwedge_F \mathbf{In}_F \rightarrow \mathcal{C}_F$.

The rules for translating constraints are given in Figure 13. The translation of the compositional, uniquess 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 $\mathbf{Final}_{cl,F}$ predicate which holds when F is not followed by a feature G which reintroduces cl and the $\mathbf{FinalIn}_{cl,F}$ predicate which further requires that F does not overwrite cl if it refines it. Subtyping constraints are represented by the $\mathbf{Sty}_{\tau,\tau'}$ variables. Truth assignments to these variables are forced to respect the class hierachy of a product specification by the final four rules in Figure 14. In effect, the STY_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 **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, WF_{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 ξ 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 $WF_{Spec} \wedge FM \rightarrow \xi$. 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 $FM \wedge WF_{Spec} \rightarrow \phi_{safe}$ 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).

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

Table 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-

$$
\begin{array}{ll}
\tau_1 \prec \tau_2 & \Rightarrow \mathbf{Sty}_{\tau_1,\tau_2} \\
\tau_2 \prec \mathbf{ftype}(\tau_1, f) & \Rightarrow \bigvee\{\mathbf{Sty}_{\tau_2,cl} \wedge \mathbf{Sty}_{\tau_1,\mathbf{type}(cld)} \wedge \mathbf{FinalIn}_{\mathbf{id}(cld),F} \mid \exists cld \in \mathbf{clds}(F), \exists cl, cl\ f \in \mathbf{fds}(cld)\}\vee \\
 & \quad \bigvee\{\mathbf{Sty}_{\tau_2,cl} \wedge \mathbf{Sty}_{\tau_1,\mathbf{type}(rcld)} \wedge \mathbf{Final}_{\mathbf{id}(rcld),F} \mid \exists rcld \in \mathbf{rclds}(F), \exists cl, cl\ f \in \mathbf{fds}(rcld)\} \\
\mathbf{ftype}(\tau_1, f) \prec \tau_2 & \Rightarrow \bigvee\{\mathbf{Sty}_{cl,\tau_2} \wedge \mathbf{Sty}_{\tau_1,\mathbf{type}(cld)} \wedge \mathbf{FinalIn}_{\mathbf{id}(cld),F} \mid \exists cld \in \mathbf{clds}(F) \exists cl, cl\ f \in \mathbf{fds}(cld)\}\vee \\
 & \quad \bigvee\{\mathbf{Sty}_{cl,\tau_2} \wedge \mathbf{Sty}_{\tau_1,\mathbf{type}(rcld)} \wedge \mathbf{Final}_{\mathbf{id}(rcld),F} \mid \exists rcld \in \mathbf{rclds}(F), \exists cl, cl\ f \in \mathbf{fds}(rcld)\} \\
\mathbf{mtype}(\tau, m) \prec \overline{\pi_k}^k \rightarrow \pi & \Rightarrow \bigvee\{\mathbf{Sty}_{cl,\pi} \wedge \bigwedge_k \mathbf{Sty}_{\pi_k,cl_k} \wedge \mathbf{FinalIn}_{\mathbf{id}(cld),F} \mid \exists cld \in \mathbf{clds}(F), \\
 & \qquad \exists cl, \overline{cl_k}^k, \overline{v_k}^k cl\ m(\overline{cl_k v_k}^k) \in \mathbf{mds}(cld)\}\vee \\
 & \quad \bigvee\{\mathbf{Sty}_{cl,\pi} \wedge \bigwedge_k \mathbf{Sty}_{\pi_k,cl_k} \wedge \mathbf{Final}_{\mathbf{id}(rcld),F} \mid \exists rcld \in \mathbf{rclds}(F), \\
 & \qquad \exists cl, \overline{cl_k}^k, \overline{v_k}^k cl\ m(\overline{cl_k v_k}^k) \in \mathbf{mds}(rcld)\} \\
\mathbf{defined}(cl) & \Rightarrow \bigvee\{\mathbf{In}_F \mid \exists cld \in \mathbf{clds}(F), \mathbf{id}(cld) = cl\} \\
\tau\ \mathbf{introduces}\ ms\ \mathbf{before}\ F & \Rightarrow \bigvee\{\mathbf{In}_G \wedge \mathbf{Prec}_{G,F} \wedge \bigwedge\{\mathbf{In}_H \rightarrow \mathbf{Prec}_{F,H} \vee \mathbf{Prec}_{H,G} \mid \exists cld' \in \mathbf{clds}(H)), \mathbf{type}(\mathbf{id}(cld')) = \tau\} \\
 & \qquad \mid \exists cld \in \mathbf{clds}(G), \mathbf{type}(\mathbf{id}(cld)) = \tau \wedge ms \in \mathbf{methods}(\mathbf{mds}(cld))\}\vee \\
 & \quad \bigvee\{\mathbf{In}_G \wedge \mathbf{Prec}_{G,F} \wedge \bigwedge\{\mathbf{In}_H \rightarrow \mathbf{Prec}_{F,H} \vee \mathbf{Prec}_{H,G} \mid \exists cld' \in \mathbf{clds}(H)), \mathbf{type}(\mathbf{id}(cld')) = \tau\} \\
 & \qquad \mid \exists rcld \in \mathbf{rclds}(G), \mathbf{type}(\mathbf{id}(rcld)) = \tau \wedge ms \in \mathbf{methods}(\mathbf{mds}(rcld))\} \\
C\ \mathbf{introduced\ before}\ F & \Rightarrow \bigvee\{\mathbf{In}_G \wedge \mathbf{Prec}_{G,F} \mid \exists cld \in \mathbf{clds}(F), \mathbf{id}(cld) = C\} \\
cl\ f\ \mathbf{unique\ in}\ C & \Rightarrow \bigwedge\{\neg\mathbf{In}_F \mid \exists cld \in \mathbf{clds}(F), \mathbf{id}(cld) = C \wedge \exists cl', cl' f \in \mathbf{fds}(cld) \wedge cl \neq cl'\}\wedge \\
 & \quad \bigwedge\{\neg\mathbf{In}_F \mid \exists rcld \in \mathbf{rclds}(F), \mathbf{id}(rcld) = C \wedge \exists cl', cl' f \in \mathbf{fds}(rcld) \wedge cl \neq cl'\} \\
cl\ m\ (\overline{vd_k}^k)\ \mathbf{unique\ in}\ C & \Rightarrow \bigwedge\{\neg\mathbf{In}_F \mid \exists cld \in \mathbf{clds}(F), \mathbf{id}(cld) = C \wedge \exists cl', \overline{vd'_k}^k, cl' m\ (\overline{vd'_k}^k) \in \mathbf{mds}(cld) \wedge cl \neq cl' \vee \\
 & \qquad (\bigvee_k vd_k \neq vd'_k)\}\wedge \\
 & \quad \bigwedge\{\neg\mathbf{In}_F | \exists rcld \in \mathbf{rclds}(F), \mathbf{id}(rcld) = C \wedge \exists cl', \overline{vd'_k}^k, cl' m\ (\overline{vd'_k}^k) \in \mathbf{mds}(rcld) \wedge cl \neq cl' \vee \\
 & \qquad (\bigvee_k vd_k \neq vd'_k)\} \\
f \notin \mathbf{fields}(\mathbf{parent}(C)) & \Rightarrow \bigwedge\{\mathbf{In}_F \wedge \mathbf{FinalIn}_{\mathbf{id}(cld),F} \rightarrow \neg\mathbf{Sty}_{\mathbf{type}(C),cl} \mid \\
 & \qquad \exists cld \in \mathbf{clds}(F), \mathbf{id}(cld) = cl \wedge C \neq cl \wedge \exists cl', cl' f \in \mathbf{fds}(cld)\}\wedge \\
 & \quad \bigwedge\{\mathbf{In}_F \wedge \mathbf{Final}_{\mathbf{id}(rcld),F} \rightarrow \neg\mathbf{Sty}_{\mathbf{type}(C),cl} \mid \\
 & \qquad \exists rcld \in \mathbf{rclds}(F), \mathbf{id}(rcld) = cl \wedge C \neq cl \wedge \exists cl', cl' f \in \mathbf{fds}(rcld)\} \\
\mathbf{pmtype}(C, m) = \tau & \Rightarrow \bigwedge\{\mathbf{In}_F \wedge \mathbf{FinalIn}_{\mathbf{id}(cld),F} \rightarrow \neg\mathbf{Sty}_{\mathbf{type}(C),cl} \mid \exists cld \in \mathbf{clds}(F), \mathbf{id}(cld) = cl \\
 & \qquad \wedge C \neq cl \wedge m \in \mathbf{methods}(cld) \wedge \mathbf{mtype}(cld, m) \neq \tau \\
 & \quad \bigwedge\{\mathbf{In}_F \wedge \mathbf{Final}_{\mathbf{id}(cld),F} \rightarrow \neg\mathbf{Sty}_{\mathbf{type}(C),cl} \mid \exists rcld \in \mathbf{rclds}(F), \mathbf{id}(rcld) = cl \\
 & \qquad \wedge C \neq cl \wedge m \in \mathbf{methods}(rcld) \wedge \mathbf{mtype}(rcld, m) \neq \tau \\
\mathbf{In}_F \Rightarrow \overline{\xi_k}^k & \Rightarrow \mathbf{In}_F \rightarrow \bigwedge_k \xi_k{}^k
\end{array}
$$

where

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

Figure 13: Translation of constraints to propositional formulas.

ule would be explicitly listed by a "requires-and-provides interface" for each module. We instead infer a module's "requires" interface automatically by considering the minimum structural requirements imposed by the the type system. We verify that these interface constraints are satisfied by the implicit "provides" interface for each feature module in a product specification. If composition is a linking process, we are guaranteeing that there will be no linking errors. The difference with normal linking is that we check all combinations of linkings allowed by the feature model.

A similar type system was proposed by Anacona et al. to type check, compile, and link source code fragments [1]. Like features, the source code fragments they considered could reference external class definitions, requiring other fragments to be included in order to build a well-typed program. These code fragments were compiled into bytecode fragments augmented with typing constraints that ranged over type variables, similar to the constraints used in the LFJ typing rules. The two approaches use these constraints for different purposes, however. Anacona et al. solve these constraints during a linking phase which combines individually compiled bytecode fragments. If all the constraints are resolved during linking, the resulting code is the same as if all the pieces had been globally compiled. Our system uses these constraints to type check a family of programs which can be built from a known set of features.

Apel et al. [3] propose a type system for Feature Featherweight Java (FFJ), a model of feature-oriented programming based on Featherweight Java [12] and prove soundness for it and some further extensions of the model. This type system is designed to check a single product specification, instead of the entire product line. Recently, the authors have extended this work to type check product lines built from FFJ [4]. A key difference between LFJ and FFJ is the use of product specifications: instead of composing a product specification to synthesize a LJ program, the FFJ semantics uses it as the final program. *g*DEEP [2] is a language-independent calculus designed to capture the core ideas of feature refinement. The type system for *g*DEEP transfers information across feature boundaries and is combined with the type system of an underlying language to type feature compositions. *g*DEEP uses a new type system which is hard to relate to existing languages, while the LFJ type system exploits the existing type system of a language to guarantee safe composition.

7. CONCLUSION

A feature model is a set of constraints describing how a set of features may be composed to build the family of programs

$$
\begin{aligned}
\textsc{Prec\_Total}: \quad & \forall A, B, A \neq B, \mathbf{In}_A \wedge \mathbf{In}_B \leftrightarrow (\mathbf{Prec}_{A,B} \vee \mathbf{Prec}_{B,A}) \\
\textsc{Prec\_ASym}: \quad & \forall A, B, \mathbf{Prec}_{A,B} \rightarrow \neg \mathbf{Prec}_{B,A} \\
\textsc{Prec\_Irrefl}: \quad & \forall A, \neg \mathbf{Prec}_{A,A} \\
\textsc{Prec\_Trans}: \quad & \forall A, B, C, (\mathbf{Prec}_{A,B} \wedge \mathbf{Prec}_{B,C}) \rightarrow \mathbf{Prec}_{A,C} \\
\textsc{Sty\_Refl}: \quad & \forall \tau, \mathbf{Sty}_{\tau,\tau} \leftrightarrow \bigvee \{\mathbf{In}_F \mid cld \in \mathbf{clds}(F) \wedge \mathbf{type}(\mathbf{id}(cld)) = \tau\} \\
\textsc{Sty\_Obj}: \quad & \mathbf{Sty}_{Object,Object} \\
\textsc{Sty\_ASym}: \quad & \forall \tau_1, \tau_2, \mathbf{Sty}_{\tau_1,\tau_2} \rightarrow \neg \mathbf{Sty}_{\tau_2,\tau_1} \\
\textsc{Sty\_Total}: \quad & \forall \tau_1, \tau_2, \tau_3, \mathbf{Sty}_{\tau_1,\tau_2} \leftrightarrow ((\mathbf{Sty}_{\tau_1,\tau_3} \wedge \mathbf{Sty}_{\tau_3,\tau_2}) \vee \\
& \bigvee \{\mathbf{In}_F \mid \exists cld \in \mathbf{clds}(F), \mathbf{type}(\mathbf{id}(cld)) = \tau_1 \wedge \mathbf{type}(\mathbf{parent}(cld)) = \tau_2\} \wedge \\
& \bigwedge \{\mathbf{In}_G \rightarrow \mathbf{Prec}_{G,F} \mid G \neq F \wedge \exists cld \in \mathbf{clds}(G), \mathbf{type}(\mathbf{id}(cld)) = \tau_1\} \wedge \\
& \bigwedge \{\mathbf{In}_G \rightarrow \mathbf{Prec}_{G,F} \mid G \neq F \wedge \exists rcld \in \mathbf{rclds}(G), \mathbf{type}(\mathbf{id}(rcld)) = \tau_1\} \vee \\
& \bigvee \{\mathbf{In}_F \mid \exists rcld \in \mathbf{rclds}(F), \mathbf{type}(\mathbf{id}(rcld)) = \tau_1 \wedge \mathbf{type}(\mathbf{parent}(cld)) = \tau_2 \wedge \mathbf{id}(rcld) \notin \\
& \mathbf{ids}(\mathbf{clds}(F))\} \wedge \\
& \bigwedge \{\mathbf{In}_G \rightarrow \mathbf{Prec}_{G,F} \mid G \neq F \wedge \exists cld \in \mathbf{clds}(G), \mathbf{type}(\mathbf{id}(cld)) = \tau_1\} \wedge \\
& \bigwedge \{\mathbf{In}_G \rightarrow \mathbf{Prec}_{G,F} \mid G \neq F \wedge \exists rcld \in \mathbf{rclds}(G), \mathbf{type}(\mathbf{id}(rcld)) = \tau_1\} \ )
\end{aligned}
$$

Figure 14: Constraints on the precedence and subtyping relations.

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 type safety of a product line by constructing SAT-instances from the interfaces of each feature. The satisfaction of the formula built from these SAT-instances ensures the product specification corresponding to the satisfying assignment will generate a well-typed LJ program. Using the feature model to guide the SAT solver, we are able to type check all the members of a product line, guaranteeing safe composition for all programs described by that feature model.

8. REFERENCES

[1] D. Ancona and S. Drossopoulou. Polymorphic bytecode: Compositional compilation for java-like languages. In *In ACM Symp. on Principles of Programming Languages 2005*. ACM Press, 2005.

[2] S. Apel and D. Hutchins. An overview of the gDEEP calculus. Technical Report MIP-0712, Department of Informatics and Mathematics, University of Passau, November 2007.

[3] S. Apel, C. Kästner, and C. Lengauer. Feature Featherweight Java: A calculus for feature-oriented programming and stepwise refinement. In *GPCE '08: Proceedings of the 7th International Conference on Generative Programming and Component Engineering*. ACM Press, Oct. 2008.

[4] S. Apel, C. Kästner, A. GröSSlinger, and C. Lengauer. Type-safe feature-oriented product lines. Technical Report MIP-0909, Department of Informatics and Mathematics, University of Passau, June 2009.

[5] D. Batory. Feature-oriented programming and the AHEAD tool suite. *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 702–703, May 2004.

[6] D. Batory. Feature models, grammars, and propositional formulas. In *In Software Product Lines Conference, LNCS 3714*, pages 7–20. Springer, 2005.

[7] A. Bergel, S. Ducasse, and O. Nierstrasz. Classbox/j: controlling the scope of change in java. In *Proc. of ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 177–189, New York, NY, USA, 2005. ACM.

[8] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer-Verlag, Berlin, 2004.

[9] K. Czarnecki and U. W. Eisenecker. Components and generative programming (invited paper). *SIGSOFT Softw. Eng. Notes*, 24(6):2–19, 1999.

[10] K. Czarnecki and K. Pietroszek. Verifying feature-based model templates against well-formedness ocl constraints. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*. ACM Press, 2006.

[11] B. Delaware, W. Cook, and D. Batory. A machine-checked model of safe composition. In *Foundations of Aspected-Oriented Languages*, 2009.

[12] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[13] R. Prieto-Diaz and J. Neighbors. Module interconnection languages: A survey. Technical report, University of California at Irvine, August 1982. ICS Technical Report 189.

[14] P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strniša. Ott: effective tool support for the working semanticist. In *ICFP '07: Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, pages 1–12, New York, NY, USA, 2007. ACM.

[15] R. Strnisa, P. Sewell, and M. J. Parkinson. The Java module system: core design and semantic definition. In R. P. Gabriel, D. F. Bacon, C. V. Lopes, and G. L. S. Jr., editors, *OOPSLA*, pages 499–514. ACM, 2007.

[16] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering*, pages 95–104, New York, NY, USA, 2007. ACM.