Foundations of Aspect-Oriented Languages (FOAL), March 2009.

A Machine-Checked Model of Safe Composition *

Benjamin Delaware, William R. Cook, Don Batory
University of Texas at Austin
{bendy,wcook,batory}@cs.utexas.edu

Abstract

Programs of a software product line can be synthesized by composing *features* which implement some 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 model

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 [4]. 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.

```
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(x);
      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

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 setBalance and update methods. 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 introduced as a static variable. Method refinement is accomplished by inheritance; Super.update(x) 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.

In addition to domain constraints, there are low-level implementation constraints that must also be satisfied. For example, a feature

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

```
feature InvestmentAccount {
  refines class Account extends WaMu {
    int 401kbalance = 0;
    refines void update (int x) {
      x = x/2; Super(); 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, RetirementAccount, and Investor features.

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

Figure 3: RetirementAccount•Bank

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 them 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 synthesizing the entire product line. 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 [11], a subset of Java that includes a formalization in the Coq proof assistant [6], using the Ott tool [10]. A program in LFJ is a set of features containing classes and class refinements. Multiple products can be constructed by selecting and composing appropriate features according to a *product specification* - a composition of features.

We define a constraint-based type system for LFJ and prove its soundness. 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 mechanically verifies that a feature model will only allow safe compositions of features, guaranteeing that the resulting programs will be type safe.

Features 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

```
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

necessary feature interfaces from the constraints generated by the LFJ type system, allowing us to check a full product line for safety without generating each product individually.

2. The Problem of Safe Composition

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

The features in Figure 2 illustrate how these modifications affect the `Account` class in the feature Bank. 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 rewrites the definition for the update method to add x to the 401k balance. InvestmentAccount also refines `Account`, updating its parent to `WaMu` and introducing a 401k field, but it refines 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 modelled as an algebra that consists of a set of operations, where each operation implements a feature. We write $M = \{$ Bank, RetirementAccount, InvestmentAccount, Investor$\}$ to mean model M has the features (operations) Bank, RetirementAccount, InvestmentAccount, Investor declared above. One or more features of a model are constants that 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 operations are unary functions on programs, and are program refinements or extensions:

InvestmentAccount•Bank	builds an investment account
RetirementAccount•Bank	builds a retirement account

where • denotes function application and $B \bullet A$ 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 defi-

nitions or modify existing definitions. The design of a program is a composition of features: 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 product line is a subset of these programs described by a feature model which constrains the ways in which features can be composed. A composition of features might fail to meet the dependencies of its constituent features, resulting in a program that fails to type check. Only a subset of the programs built from a set of features is well-typed. The goal of safe composition is to ensure that the product line described by a feature model is contained in this set, i.e. that all the programs in the product line are well-typed.

The combinatorial nature of product lines presents a number of problems to statically 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. Being a forward-thinking investor, the employee adds this sum 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 [11]. 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 [8] because of its treatment of constructors. When composing features, it is important to be able to add new member variables to a class during refinement. Featherweight Java requires all member variables to be initialized in a single constructor call. As a result, adding a new member variable causes all previous constructor calls to be invalid. Lightweight Java allows such refinements through its support of more flexible initialization of member variables. In addition, Lightweight Java has a full formalization in Coq, which we extended to prove the soundness of LFJ mechanically. The proof scripts for the system are available at `http://www.cs.utexas.edu/~bendy/featurejava.php`.

The syntax of LFJ extends LJ to support feature-oriented programming is given in Figure 6. A feature definition FD maps a feature name F to a list of class declarations $\overline{cld}$ and a list of class

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

Figure 6: Modified Syntax of Lightweight Feature Java.

refinements $\overline{rcld}$. A class refinement $rcld$ includes a class name dcl, 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 set of features from which a product line can be built is called the *feature table*. A product specification PS is a sequence of distinct feature names.

3.1 Feature Composition

A LJ program can be modelled as a partial function from class names to their definitions: $CT : dcl \rightarrow cld$. In the operational semantics of LJ, this function is concretely realized as the function $\mathbf{path} : P \rightarrow dcl \rightarrow cld$ which looks up a class definition in a given program. In this context, CT is simply the $\mathbf{path}$ specialized on P: $CT = \mathbf{path}_P$. Features are themselves functions from LJ programs to LJ programs. Composition of a feature **feature** `FD` $\{\overline{\texttt{cld}}; \overline{\texttt{rcld}}\}$ with an LJ program P produces a new mapping, CT':

$$CT'(dcl) = \begin{cases} \mathbf{path}_{\overline{cld}}(dcl) & dcl \in \overline{cld} \\ \overline{rcld} \bullet CT(dcl) & dcl \notin \overline{cld} \end{cases} \quad (1)$$

In the case that `FD` introduces a class named dcl, CT' returns this class, ignoring any previous declarations and refinements of that class. Otherwise, CT' finds the definition of dcl in the previous program using the original CT function and returns the resulting class definition, cld, refined by $\overline{rcld}$. If a class refinement $rcld$ in $\overline{rcld}$ is named dcl, the $\bullet$ operator builds a refined class by first advising the methods of cld with the method refinements in $rcld$. The fields and methods introduced by $rcld$ are then added to this class and the parent of the resulting class is set to the superclass named in $rcld$. Refinement fails if cld lacks a method with a signature refined by $rcld$.

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

3.2 Safe Composition

A feature model is safe if it only allows the creation of well-formed LJ programs. For any particular product specification, this can be checked by composing the specification and then checking the safety of the resulting program using the standard LJ type sys-

tem. 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 considers a potentially exponential number of programs, making it a computationally expensive process. Instead, we propose a type system that can 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 CT function. For example, CT is used to determine that the type of a variable y is a subtype of the type of variable x when assigning y to x in a method body. 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. For a feature in a given feature table, this interface determines which features must be included with it in a product specification in order for its constructs to be well-formed in the final LJ program.

4. LFJ Type System

We present a constraint-based type system for LFJ based on a constraint-based type system we have developed for LJ. The constraint-based 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 their satisfaction. Generating a set of constraints for a feature is separated from consideration of which product specifications have a combination of features satisfying these constraints. The constraints used by

Composition Constraints

dcl **introduces** ms **before** F
dcl **introduced before** F

Uniqueness Constraints

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

Structural Constraints

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

Figure 7: Syntax of Lightweight Feature Java typing constraints.

our type system are given in Figure 7 and are divided into three 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 a class named dcl, a restriction in the LJ formalization. The structural constraints come from the standard LJ type system and constrain the set of members of a class and the class hierarchy in the final program. The subtype constraint is particularly important because the class hierarchy is malleable until composition time; if it were static, constraints that depend on subtyping class could be reduced to other constraints or eliminated entirely.

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

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

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

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

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

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

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

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

Figure 8: Typing Rules for LJ and LFJ statements.

The typing rules for LFJ are found in Figures 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, called a *signature*. The signature ξ provides an explicit interface which guarantees that J holds in any product specification that satisfies ξ. Typing rules for statements, methods, and classes are those from LJ augmented with signatures. 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 which already performs these checks must precede the refinement in order for it to be well-formed. The signature of a product specification PS is the union of the constraints on each of the features in PS.

Once the signature of a product specification PS is generated according to the rules in Figure 10, we evaluate whether it is satisfied by PS using the rules in Figure 11. Compositional constraints on a feature F are satisfied when a feature with the appropriate introductions precedes F in PS. Uniqueness constraints are satisfied

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

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

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

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

$\boxed{\vdash_{\mathtt{F}} rcld \mid \mathcal{C}}$ Class refinement well-formed in feature with constraints

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

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

when no two features in PS introduce a member with the same name but different signatures to a class dcl. In LFJ, satisfaction of structural constraints is evaluated as in LJ, replacing uses of **path** with the CT function built by composition of the features in PS.

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

4.1 Soundness of the LFJ Type System

The soundness proof is based on successive refinements of the type systems of LJ and LFJ, reducing them to the proofs of progress and preservation of the original LJ type system given in [11]. We first use our constraint-based type system for LJ, utilizing the structural constraints listed in Figure 7 and the corresponding judgements in Figure 11 to check constraint satisfaction. This type system is shown to be equivalent to the original LJ type system, in that a program with unique class names and an acyclic class hierarchy satisfies its signature if and only if it is well-formed according to the original typing rules. We then show that if a single LFJ product specification is well-formed according to the constraint-based LFJ type system, it produces a LJ program that is also well-formed. 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 proofs of soundness.

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

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

$$\frac{\overline{\vdash cld_k \mid \mathcal{C}_k}^{k} \qquad P = \overline{cld_k}^{k} \quad \textbf{distinct names}\ (P)}{\vdash P \mid \bigcup_k \mathcal{C}_k} \qquad \text{(WF-PROGRAM)}$$

$\boxed{\vdash \texttt{F} \mid \mathcal{C}}$ Feature well-formed with constraints

$$\frac{\overline{\vdash cld_k \mid \mathcal{C}_k}^{k} \qquad \overline{\vdash_F rcld_\ell \mid \mathcal{C}_\ell}^{\ell}}{\vdash \textbf{feature}\ \texttt{F}\ \{\overline{cld_k}^{k}\ \overline{rcld_\ell}^{\ell}\} \mid \bigcup_k \mathcal{C}_k \cup \bigcup_\ell \mathcal{C}_\ell} \qquad \text{(WF-FEATURE)}$$

$\boxed{\vdash \texttt{PS} \mid \mathcal{C}}$ Product specification well-formed with constraints

$$\vdash \emptyset \mid \emptyset \qquad \text{(WF-SPECIFICATION-NIL)}$$

$$\frac{\vdash F \mid \mathcal{C} \qquad \vdash \overline{F_k}^{k} \mid \mathcal{C}'}{\vdash F, \overline{F_k}^{k} \mid \mathcal{C} \cup \mathcal{C}'} \qquad \text{(WF-SPECIFICATION)}$$

Figure 10: Typing Rules for LFJ Programs and Features.

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 Fig. 9 and the 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 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 appears as a constraint in the signature. As a result of the previous argument, satisfaction of the signature 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 the LFJ Type System)**.** *Let PS be a LFJ product specification and $\mathcal{C}$ be a set of constraints such that $\vdash PS \mid \mathcal{C}$. If $PS \models \mathcal{C}$ and **Object** is in the path of every class introduced by a feature in PS, then the composition of the features in PS produces a valid, well-formed LJ program.*

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

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

For each class or method refinement of a feature F in PS, a composition constraint is generated by the LFJ typing rules. Each of these are satisfied according to the definition in Fig. 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 of PS will succeed.

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

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

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

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

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

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

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

$$\frac{FP = \overline{A_k}^{k} F \overline{B_\ell}^{\ell} H \overline{C_j}^{j} \qquad \tau.ms \in H \qquad \tau \notin \textbf{introductions}(\overline{B_\ell}^{\ell})}{FP \models \tau\ \textbf{introduces}\ ms\ \textbf{before}\ F}$$

$$\frac{FP = \overline{A_k}^{k} F \overline{B_\ell}^{\ell} H \overline{C_j}^{j} \qquad \textbf{dcl} \in H}{FP \models dcl\ \textbf{introduced before}\ F}$$

$$\frac{\textbf{type}(dcl) = \tau \qquad \forall A, B \in FP, \tau.cl_1\ f \in A \wedge \tau.cl_2\ f \in B \to cl_1 = cl_2}{FP \models cl\ f\ \textbf{unique in}\ dcl}$$

$$\frac{\textbf{type}(dcl) = \tau \qquad ms_1 = cl\ m\ (\overline{vd_k}^{k}) \qquad ms_2 = cl'\ m\ (\overline{vd'_k}^{k}) \qquad \forall A, B \in FP, \tau.ms_1 \in A \tau.ms_2 \in B \to ms_1 = ms_2}{FP \models cl\ m\ (\overline{vd_k}^{k})\ \textbf{unique in}\ dcl}$$

Figure 11: Satisfaction of typing constraints.

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

In essence, we must show that the premises of the constraint-based LJ typing judgements hold. Our assumption that each class in PS is a descendant of **Object** ensures that P has an acyclic, well-founded class hierarchy. The premises for the LJ methods and statements are identical, leaving class typing rules for us to consider. The LJ typing rules require that the method and field names for a class be distinct, but these premises are removed by the LFJ typing rules, as the members of a class are not finalized until after composition. This requirement is instead enforced by the uniqueness constraints in Fig. 11, which are satisfied only when a method or field name is 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 some set of constraints $\mathcal{C}'$ such that $\vdash P \mid \mathcal{C}'$.

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

We break this proof into two sublemmas:

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

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

All constraints in $\mathcal{C}'$ appear in $\mathcal{C}$, so $PS \models \mathcal{C}'$. By (b) above, it follows that $P \models \mathcal{C}'$. P must therefore be a well-formed LJ program by Theorem 4.1.

□

5. Feature Models

A *feature model* represents the dependencies and constraints between features that make up a product line. One common representation for feature models is a *feature diagram*. A feature diagram is a hierarchy of features where each node in the tree corresponds to a feature. Annotations on the tree represent constraints. Features required by a parent are marked with a dot.

5.1 Feature Diagrams

Consider an elementary automotive product line that differentiates cars by transmission type (automatic or manual), engine type (electric or gasoline), and the option of cruise control. Figure 12 shows the feature diagram of this product line. A car has a body, engine, transmission, and optionally a cruise control. A transmission is either automatic or manual (choose one), and an engine is electric-powered, gasoline-powered, or both.

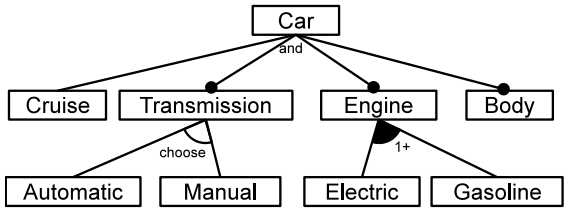

Figure 12: Feature diagram

Besides hierarchical relationships, feature models also allow cross-tree constraints, although these are more difficult to represent in a feature diagram. Such constraints are often inclusion or exclusion statements of the form: if feature F is included in a product, then features A and B must also be included (or excluded). A cross-tree constraint is that cruise control requires an automatic transmission.

Feature models are compact representations of propositional formulas [5]. We exploit this representation in relating feature models to the constraint-based type system for LFJ.

5.2 Propositional Representation of Feature Models

A feature model determines the set of legal combinations of features in the algebra that defines product lines. A given program specification can be tested for validity by checking if it satisfies the constraints expressed in a feature model. For example, the feature model *Auto* of the automotive product line is:

(Body ∧ (Automatic ∨ Manual) ∧
(Electric ∨ Gasoline) ∧ (Automatic ↔ ¬ Manual))

where Body is the lone constant. Some products (i.e. legal expressions or sentences) of this product line are:

c1 = Automatic•Electric•Body
c2 = Cruise•Automatic•Electric•Gasoline•Body

$c1$ is a car with an electric engine and automatic transmission. $c2$ is a car with both electric and gasoline engines, automatic transmission, and cruise control.

6. Safe Composition for Feature Models

By the soundness of the LFJ type system, the satisfaction of the signature of every feature in a product specification is sufficient to guarantee that its composition is a well-formed program. The signature of a feature F provides an interface with other feature modules. This interface can be translated into a propositional formula describing the minimal structural requirements that any product specification built from a feature table FT which includes F must satisfy in order for the constructs in F to be well-formed. The conjunction of these formulas builds a formula ϕ_{safe} which any product specification must satisfy in order to produce a well-formed program. The safety of a feature model can then be statically verified by using a SAT solver to check that its propositional representation implies this minimal formula.

$\mathbf{In}_A$: 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 13: Description of propositional variables.

The propositional variables of ϕ_{safe} have three basic forms, described in Figure 13. Note that a satisfying assignment to the **In** and **Prec** variables describes a unique product specification as long as it obeys the properties of the precedence relations. The propositional constraints that impose these properties are given in 14. The first three formulas enforce that a precedence relation is total on all features included in a product specification, that it is asymmetric, and that it is irreflexive. The next four constraints ensure that each product specification dictates an assignment to the **Sty** variables corresponding to its class hierarchy. 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; this mirrors the construction of the subtyping relation used in the original LJ type system. A satisfying assignment to WF_{Spec}, the conjunction of all these constraints, represents a unique product specification.

The makeup of the program built from a product specification depends upon the ordering of features and their introductions and refinements. The rules in Figure 15 generate a propositional formula for each kind of typing constraint. A satisfying assignment to a formula in Figure 15 which also satisfies WF_{Spec} represents a product specification which satisfies the associated con-

$$
\begin{array}{rl}
\textsc{Prec\_Total}: & \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}: & \forall A, B, \mathbf{Prec}_{A,B} \rightarrow \neg \mathbf{Prec}_{B,A} \\
\textsc{Prec\_Irrefl}: & \forall A, \neg \mathbf{Prec}_{A,A} \\
\textsc{Sty\_Refl}: & \forall \tau, \mathbf{Sty}_{\tau,\tau} \leftrightarrow \bigvee \{\mathbf{In}_F \mid cld \in \mathbf{clds}(F) \wedge \mathbf{type}(\mathbf{name}(cld)) = \tau\} \\
\textsc{Sty\_Obj}: & \mathbf{Sty}_{\texttt{Object},\texttt{Object}} \\
\textsc{Sty\_ASym}: & \forall \tau_1, \tau_2, \mathbf{Sty}_{\tau_1,\tau_2} \rightarrow \neg \mathbf{Sty}_{\tau_2,\tau_1} \\
\textsc{Sty\_Total}: & \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{name}(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{name}(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{name}(rcld)) = \tau_1\} \vee \\
& \bigvee \{\mathbf{In}_F \mid \exists rcld \in \mathbf{rclds}(F), \mathbf{type}(\mathbf{name}(rcld)) = \tau_1 \wedge \mathbf{type}(\mathbf{parent}(cld)) = \tau_2 \wedge \\
& \mathbf{name}(rcld) \notin \mathbf{names}(\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{name}(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{name}(rcld)) = \tau_1\}\ ) \\
\textsc{Sty\_WF}: & \forall A, \forall c \in \mathbf{clds}(A), \mathbf{In}_A \rightarrow \mathbf{Sty}_{\mathbf{ty}(\mathbf{name}(c)),\texttt{Object}}
\end{array}
$$

Figure 14: Constraints on the precedence and subtyping relations.

straint. The **Final** and **FinalIn** abbreviations ensure that introductions and refinements in features appearing before the feature with the final introduction are ignored. The composition and uniqueness constraints have straightforward propositional representations that govern the valid orderings and makeup of product lines. The translations of the structural constraints rely on the mutability of the class hierarchy; since the class hierarchy of the product line is flexible, any class cl_1 that has a required field or method could ultimately satisfy a constraint on the members of another class, cl_2, if $cl_2 \prec cl_1$ in the final product specification.

Let ϕ_F be the conjunction of the formulas built from each constraint in the signature of a feature F according to the rules in Figure 15. ϕ_F describes the structure of all product specifications in which F is well-formed. ϕ_{safe} is constructed by first building a clause for each feature F stating its inclusion implies ϕ_F: $\mathbf{In}_F \rightarrow \phi_F$. The propositional constraints generated by STY_WF in Figure 14 are then added to this formula to ensure that the class hierarchy of a product specification is acyclic by requiring that each class included in a product specification be a subtype of `Object`.

The representation of a feature model in propositional logic, FM, describes the assignments that represent legitimate specifications of a product line, defining the family of programs it contains. It is possible to build FM using the variables in Figure 13. By construction, a satisfying assignment to ϕ_{safe} which sets $\mathbf{In}_F$ to true also satisfies ϕ_F. It follows that any satisfying assignment to $WF_{Spec} \rightarrow \phi_{safe}$ represents a product specification which satisfies the signatures of each of the features in it. By Theorem 4.2, such a product specification produces a well-formed LJ program. Since FM and the minimal well-formedness formula share the same variables, a SAT solver can check whether $FM \wedge WF_{Spec} \rightarrow \phi_{safe}$ is valid. If so, the set of programs described by the feature model is a subset of those allowed by ϕ_{safe}. Thus, the composition of any product specification which satisfies the syntactic constraints of such a feature model is well-formed.

6.1 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 this approach for checking safe composition of AHEAD software product lines [12]. 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.

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].

7. Related Work

Our strategy of representing feature models as propositional formulas in order to verify their consistency was first proposed in [5]. 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 [7] 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 [5], 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 for our current approach was a system developed by Thaker et al. [12] 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 the feature models of these product lines. 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.

If features are thought of as modules, the feature model used to describe a product line is a *module interconnection language* [9]. Normally, the typing requirements for a module would be

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

where

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

Figure 15: Translation of constraints to propositional formulas.

explicitly listed by a "requires-and-provides interface" for each module. We infer this interface automatically by considering the minimum structural rules required of a feature module by the type system. We verify that these interface constraints are satisfied by the implicit interface given to each module by the feature module. 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.

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. [3] propose a type system for a model of feature-oriented programming based on Featherweight Java [8] and prove soundness for it and some further extensions of the model. *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 for an underlying language to type feature compositions.

8. Conclusion

A feature model is a set of constraints describing how a set of features may be composed to build the family of programs in a product line. This feature model is safe if it only allows the construction of well-formed programs. Simply iterating all the programs described by the feature model is computationally expensive and impractical for large product lines. In order to verify statically 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 for 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.

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*, pages 26–37. ACM Press, 2005.

[2] S. Apel and D. Hutchins. An overview of the gDEEP calculus. Technical Report 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 *Proceedings of the 7th International Conference on Generative Programming and Component Engineering (GPCE)*. ACM Press, Oct. 2008.

[4] 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.

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

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

[7] 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*, pages 211–220, New York, NY, USA, 2006. ACM.

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

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

[10] 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.

[11] 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.

[12] 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.