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

Benjamin Delaware
William Cook
Don Batory
University of Texas at Austin
Safe Composition

- **Features**
  - Word Processor has formatting, printing, spell check, tables...
  - Cut across traditional modularity boundaries
  - Reify functionality into distinct feature modules

- **Software Product Line (SPL)**
  - Multiple products from one code base
  - Product = subset of features

- **Safe Composition**
  - Type check all products
  - Products are exponential in number of features

- **Goal**
  - Sound type system
  - Foundation for efficient implementation
A Feature Example

• Features are sets of class definitions and refinements
A Feature Example

- Features are sets of class definitions and refinements

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

- Features are sets of class definitions and refinements

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

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

feature RetireAccount {
  refines class Account extends Lehman {
    int 401kbalance = 10000;
    refines void update (int x) {
      401kbalance += x;
    }
  }
}
```
Composing Features

- Features are sets of class definitions and refinements

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

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

feature RetireAccount {
    refines class Account extends Lehman {
        int 401kbalance = 10000;
        int update (int x) {
            401kbalance += x;
        }
    }
}
```
Composing Features

Account
InvestAccount
RetireAccount
class Account extends Object {
    int balance = 0;
    void update(int x) {
        int newBal = balance + x;
        balance = newBal;
    }
}

InvestAccount • Investor
InvestAccount • Investor

Account
InvestAccount

Account

InvestAccount

RetireAccount

**Account**

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

**InvestAccount**

```java
feature InvestAccount {
    refinest class Account extends WaMu {
        int 401kbalance = 0;
        refinest void update(int x) {
            x = x/2;
            Super();
            401kbalance += x;
        }
    }
}
```
InvestAccount • Investor

Account + InvestAccount =

```java
class Account extends WaMu {
    int balance = 0;
    int 401kbalance = 0;
    void update(int x) {
        int newBal = balance + x;
        balance = newBal;
        401kbalance += x;
    }
}
```

RetireAccount
InvestAccount • Investor

InvestAccount

RetireAccount

Account

= Account

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

Account
InvestAccount

RetireAccount

Account

RetireAccount

InvestAccount

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

RetireAccount

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

Account
RetireAccount \cdot \text{Investor}
Feature Models

• A SPL has a set of available features:
  \{Account, RetireAccount, InvestAccount\}

• Typically feature combinations are constrained
  • A feature model represents these constraints
  • Propositional formula is compact representation [Batory05]
    \( RetireAccount \lor InvestAccount \)

• Product corresponds to truth assignment

• FMs should enforce implementation constraints
  • Safe Composition
    \((RetireAccount \lor InvestAccount) \land (RetireAccount \Rightarrow Account) \land (InvestAccount \Rightarrow Account)\)
Checking Safe Composition

• Could synthesize entire product line
  • Computationally expensive:
Checking Safe Composition

• Could synthesize entire product line
• Computationally expensive:

Account  InvestAccount  RetireAccount  Bailout  Employer  NYSE
Checking Safe Composition

- Could synthesize entire product line
- Computationally expensive:
Difficulties

• Combinatorial nature of SPLs problematic:

```java
feature Payroll {
    class Employer extends Object {
        Account Employee1;
        ...
        Employee1.401kbalance += 10000;
        ...
    }
}
```

• Bailout feature needs Account
  • Account needs 401kbalance
  • Multiple ways to satisfy
  • Introduction
  • Inheritance

• Features are static
  • Surrounding program is not

• Dependencies are resolved by a combination of features
  • These features have their own set of dependencies
Lightweight Feature Java

- Lightweight Java [Strnisa07]
  - Minimal imperative subset of Java formalized in Coq
- Lightweight Feature Java
  - Lightweight Java extended with features

Feature Table

\[ FT ::= \{ FD \} \]

Product specification

\[ PS ::= F \]

Feature declaration

\[ FD ::= \text{feature } F \{ cld; rcld \} \]

Class refinement

\[ rcld ::= \text{refines class } dcl \text{ extending } cl \{ fd; md; rmd \} \]

Method Refinement

\[ rmd ::= \text{refines method } ms \{ s; \text{Super}(); s; \text{return } y \} \]

- Formalized in the Coq Proof Assistant
Composition in LFJ

- Programs built from product specifications

- **compose**
  - Refine existing classes
    - Apply method refinement
    - Introduce fields, methods
  - Introduce new classes

- Recursively apply **compose** to specification
A unique feature within the method ensures the uniqueness of signatures.

\[
\text{distinct}(\overline{\text{var}_k^k})
\]

\[
\text{type}(cl_k) = \tau_k^k
\]

\[
\text{type}(cl) = \tau'
\]

\[
\Gamma = [\overline{\text{var}_k^k}^{\prime \prime} \tau_k^k][\text{this} \sp{\prime \prime} \tau]
\]

\[
\Gamma(y) = \tau''
\]

\[
\mathcal{P}, \Gamma \| s_\ell
\]

\[
\mathcal{P} \vdash \tau'' \quad \tau'
\]

\[
\mathcal{P} \vdash \text{defined } cl_k^k
\]

\[
\mathcal{P} \|_{\tau} \text{cl meth } (\overline{cl_k \text{var}_k^k}) \{s_\ell \text{ return } y; \}
\]

- Program not available until composition
LJ Type System

\[
\text{distinct}(\var_k^k) \\
\frac{\text{type}(cl_k) = \tau_k}{\text{type}(cl) = \tau'} \\
\Gamma = [\var_k^k \tau_k^k][\text{this } \tau] \\
\Gamma(y) = \tau'' \\
P, \Gamma \vdash \ell \\
P \vdash \tau'' \quad \tau' \\
P \vdash \text{defined } cl_k^k \\
P \vdash_{\tau} cl \text{ meth } (cl_k \var_k^k) \{s_{\ell} \text{ return } y;\}
\]

Internal Checks

(WF-METHOD)

- Program not available until composition


LJ Type System

distinct(var_{k}^{k})

type(cl_{k}) = \tau_{k}^{k}

type(cl) = \tau'

\[ \Gamma = [ \frac{var_{k}^{k}}{\tau_{k}^{k}} ] [ \text{this} \quad \tau ] \]

\[ \Gamma(y) = \tau'' \]

P, \Gamma \vdash s_{\ell}^{\ell}

P \vdash \tau'' \quad \tau'

P \vdash \text{defined} \ cl_{k}^{k}

---

P \vdash_{\tau} cl \ meth (cl_{k} \ var_{k}^{k}) \ {s_{\ell}^{\ell} \ return \ y; }{n}

• Program not available until composition
• External premises become constraints

\[ \text{distinct}(\overline{\text{var}}_k^k) \]
\[ \text{type}(cl_k) = \tau_k \]
\[ \text{type}(cl) = \tau' \]
\[ \Gamma = [\overline{\text{var}}_k^" \tau_k^k][\text{this } \tau] \]
\[ \check{\Gamma}(s_\ell | C_\ell) \]
\[ \frac{\Gamma(y) = \tau''}{\tau \text{ cl meth } (cl_k \overline{\text{var}}_k^k) \{s_\ell \text{ return } y;\} \mid \{\tau'' \tau', \text{ defined } cl_k^k\} \# \bigcup_\ell C_\ell} \]

• Compositional Constraints
• Uniqueness Constraints
• Structural Constraints
Constraint-Based Typing

- Two typing phases
- Typing Feature Tables

\[ \vdash FD_k \mid WF_k \]
\[ \vdash \{FD_k\} \mid \bigcup_k \{In_{FD_k} \Rightarrow WF_k\} \]

- Well-typed product specification

\[ PS \vdash \bigcup_k \{In_{FD_k} \Rightarrow WF_k\} \]

- Feature Constraint
- Compositional Constraints
- Uniqueness Constraints
- Structural Constraints
Soundness of LFJ Type System

**Theorem:**
\[ \vdash \{\text{FD}_k\} \mid \bigcup_k \{\text{In}_{\text{FD}_k} \Rightarrow \text{WF}_k\} \]
\[ \text{PS} \vdash \bigcup_k \{\text{In}_{\text{FD}_k} \Rightarrow \text{WF}_k\} \]
\[ \vdash_{\text{FJ}} \text{compose}(\text{PS}) \]

- **Space of products**

- First premise describes subset of type-safe products
- Second ensures product in this space
Soundness of LFJ Type System

**Theorem:**

\[
\vdash \{\text{FD}_k\} \mid \bigcup_k \{\text{In}_{\text{FD}_k} \Rightarrow \text{WF}_k\}
\]

\[
\text{PS} \vdash \bigcup_k \{\text{In}_{\text{FD}_k} \Rightarrow \text{WF}_k\}
\]

\[
\vdash_{\text{FJ}} \text{compose}(\text{PS})
\]

- Space of products
- First premise describes subset of type-safe products
- Second ensures product in this space
Soundness of LFJ Type System

**Theorem:**

\[ \vdash \{ \text{FD}_k \} \mid \bigcup_k \{ \text{In}_{\text{FD}_k} \Rightarrow \text{WF}_k \} \]

\[ \text{PS} \vdash \bigcup_k \{ \text{In}_{\text{FD}_k} \Rightarrow \text{WF}_k \} \]

\[ \vdash_{FJ} \text{compose}(\text{PS}) \]

- Space of products
- First premise describes subset of type-safe products
- Second ensures product in this space
Validating Feature Models

- Feature Models describe desired product space
  - Should be contained in type-safe space
- Recall Feature Models are propositional formulas
  - Describe type-safe space in propositional logic, $WF_{\text{Safe}}$
  - Reduction from typing constraints
- Reduce to SAT:

$$FM \Rightarrow WF_{\text{Safe}}$$
Validating Feature Models

- Feature Models describe desired product space
  - Should be contained in type-safe space
- Recall Feature Models are propositional formulas
  - Describe type-safe space in propositional logic, $WF_{\text{Safe}}$
  - Reduction from typing constraints
- Reduce to SAT:
  \[ FM \Rightarrow WF_{\text{Safe}} \]
• Feature Models describe desired product space
  • Should be contained in type-safe space
• Recall Feature Models are propositional formulas
  • Describe type-safe space in propositional logic, $WF_{\text{Safe}}$
  • Reduction from typing constraints
• Reduce to SAT:
  \[ FM \Rightarrow WF_{\text{Safe}} \]
Evaluation

- Checking validity coNP-complete in general
- Our formulas are highly structured

<table>
<thead>
<tr>
<th>Product Line</th>
<th># of Features</th>
<th># of Programs</th>
<th>Code Base Jak/Java LOC</th>
<th>Program Jak/Java LOC</th>
<th>Typechecking Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>JPL</td>
<td>70</td>
<td>56</td>
<td>34K/48K</td>
<td>22K/35K</td>
<td>&lt;30s</td>
</tr>
</tbody>
</table>

- Previous implementation of approach [Thaker07]
  - Identified errors in existing product lines
- Evidence of erroneous product
Conclusion

- Feature-based Software Product Lines
- Safe Composition
- Lightweight Feature Java
  - Verified in Coq proof assistant
  - Constraints describe program space
- Validating Feature Models
  - Reduce to SAT
  - Efficient evaluation
Questions?