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
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;
    }
  }
}
```

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

```java
feature RetireAccount {
  refines class Account extends Lehman {
    int 401kbalance = 10000;
    refines int 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;
        }
    }
}
```

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

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

Account
InvestAccount
RetireAccount
InvestAccount • Investor

Account  InvestAccount  RetireAccount
**InvestAccount • Investor**

```
InvestAccount
RetireAccount

Account

feature Account {
  class Account extends Object {
    int balance = 0;
    void update(int x) {
      int newBal = balance + x;
      balance = newBal;
    }
  }
}
```
```java
// InvestAccount Feature
feature InvestAccount {
  refines class Account extends WaMu {
    int 401kBalance = 0;
    refines void update (int x) {
      x = x/2;
      Super();
      401kBalance += x;
    }
  }
}

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

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

Account
InvestAccount

RetireAccount • Investor

 InvestAccount

feature Account {
    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 401kBbalance = 10000;
        int update (int x) {
            401kBbalance += x;
        }
    }
}

Account

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

RetireAccount

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

Account

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

RetireAccount

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

Account
InvestAccount

RetireAccount • Investor

Account + RetireAccount = InvestAccount

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

Account
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
    \[(RetireAccount \lor InvestAccount) \land (RetireAccount \Rightarrow Account) \land (InvestAccount \Rightarrow Account)\]
• 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:

```
Account  InvestAccount  RetireAccount  Bailout  Employer  NYSE
```

```
Account
Account  InvestAccount
Account  RetireAccount
Account  Bailout
Account  InvestAccount  Bailout
Account  RetireAccount  Bailout
Account  Bailout  Employer
Account  InvestAccount  Bailout  Employer
Account  RetireAccount  Bailout  Employer
Account  Bailout  Employer  NYSE
```

...
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

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

Product specification

\[ \text{PS} ::= F \]

Feature declaration

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

Class refinement

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

Method Refinement

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

- 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
**LJ Type System**

- Program not available until composition
### LJ Type System

#### Internal Checks

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[\text{distinct}(\overline{\text{var}_k^{\ell}})]</td>
<td>ertz</td>
</tr>
<tr>
<td>[\text{type}(\text{cl}_k) = \tau_k^{\ell}]</td>
<td>ertz</td>
</tr>
<tr>
<td>[\text{type}(\text{cl}) = \tau']</td>
<td>ertz</td>
</tr>
<tr>
<td>[\Gamma = [\overline{\text{var}_k \mapsto \tau_k^{\ell}}][\text{this} \mapsto \tau]]</td>
<td>ertz</td>
</tr>
<tr>
<td>[\Gamma(y) = \tau'']</td>
<td>ertz</td>
</tr>
<tr>
<td>[\text{P}, \Gamma \vdash s_{\ell}]</td>
<td>ertz</td>
</tr>
<tr>
<td>[\text{P} \vdash \tau'' \prec \tau']</td>
<td>ertz</td>
</tr>
<tr>
<td>[\text{P} \vdash \text{defined} \overline{\text{cl}_k^{\ell}}]</td>
<td>ertz</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[\text{P} \vdash \tau \text{cl meth} (\overline{\text{cl}_k \text{var}<em>k^{\ell}}) {s</em>{\ell} \text{return } y; }]</td>
<td>ertz</td>
</tr>
</tbody>
</table>

- Program not available until composition
### LJ Type System

**distinct**($\overline{\text{var}_k}$)

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

\[
\Gamma = [\overline{\text{var}_k} \mapsto \tau_k] [\text{this} \mapsto \tau]
\]

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

\[
P, \Gamma \vdash s^{\ell}_l
\]

\[
P \vdash \tau'' < \tau'
\]

\[
P \vdash \text{defined} \ cl_k
\]

\[
P \vdash \tau \ \text{cl method} \ (\overline{cl_k \ \text{var}_k}) \ \{s^{\ell}_l \ \text{return} \ y; \}
\]

**External Checks**

**WF-Method**

- Program not available until composition
Constraint-Based Typing

- External premises become constraints

\[
\begin{aligned}
\text{distinct}(\var_k^k) \\
type(cl_k) = \tau_k^k \\
type(cl) = \tau' \\
\Gamma = [\var_k^k \mapsto \tau_k^k][\text{this} \mapsto \tau] \\
\Gamma \vdash s_\ell \mid C_\ell^\ell \\
\Gamma(y) = \tau''
\end{aligned}
\]

\(\vdash_{\tau} cl \ \text{meth } (cl_k \ \var_k^k) \ \{s_\ell^\ell \ \text{return y;} \} \mid \{\tau'' \prec \tau', \text{defined } cl_k^k\} \cup \bigcup_\ell C_\ell\)

- 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:**

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

- Space of products
- First premise describes subset of type-safe products
- Second ensures product in this space
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
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, $WF_{Safe}$
- Reduction from typing constraints

Reduce to SAT:
$$FM \Rightarrow WF_{Safe}$$
• 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, $\mathbf{WF}_{\text{Safe}}$
  • Reduction from typing constraints

• Reduce to SAT:
  $$\mathbf{FM} \Rightarrow \mathbf{WF}_{\text{Safe}}$$
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?