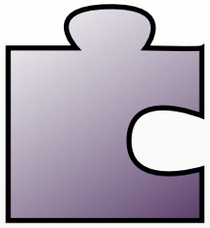
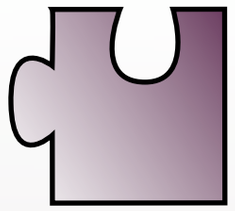
Fitting the Pieces Together: A Machine-Checked Model of Safe Composition

Benjamin Delaware
William Cook
Don Batory
University of Texas at Austin

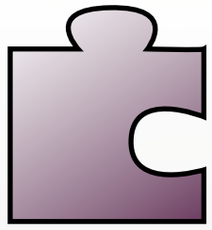


Safe Composition

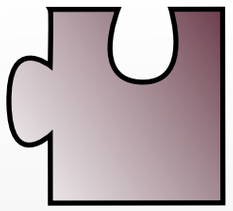


- **Features**
  - Word Processor has formatting, printing, spell check, tables..
  - Cut across traditional modularity boundaries
  - Reify functionality into distinct **feature modules**
- **Software Product Line (SPL)**
  - Multiple products from one code base
  - Product = subset of features
- **Safe 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

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

Account

A Feature Example

- Features are sets of class definitions and refinements

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

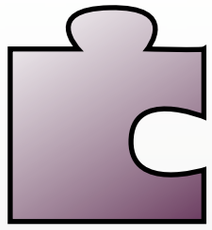
Account

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

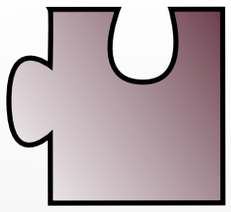
InvestAccount

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

RetireAccount



Composing Features



- Features are sets of class definitions and refinements

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

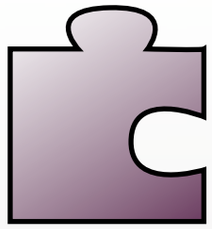
Account

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

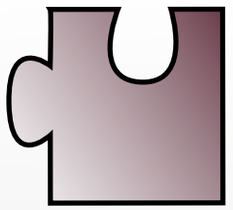
InvestAccount

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

RetireAccount



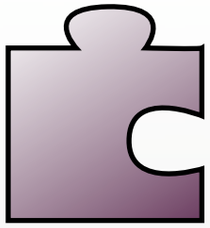
Composing Features



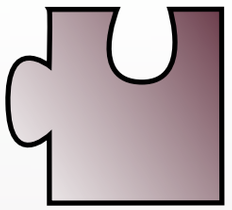
Account

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;
    }
  }}

Account

InvestAccount • Investor

RetireAccount

Account

InvestAccount

=

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

InvestAccount

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

Account

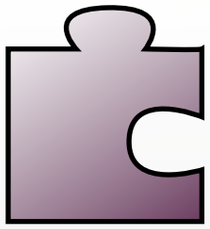
InvestAccount • Investor

RetireAccount

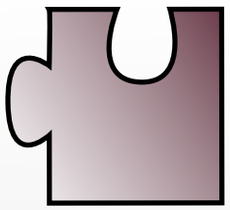
Account + InvestAccount =

```
feature Account {
  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;
    }
}}
```

Account



RetireAccount • Investor



Account

InvestAccount

RetireAccount

RetireAccount • Investor

InvestAccount

RetireAccount

Account

=

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

Account

RetireAccount • Investor

InvestAccount

Account

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 401kbalance = 10000;
     int update (int x) {
      401kbalance += x;
    }
  }}
```

RetireAccount

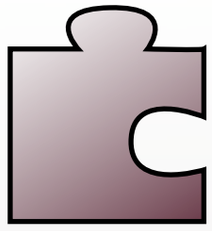
RetireAccount • Investor

InvestAccount

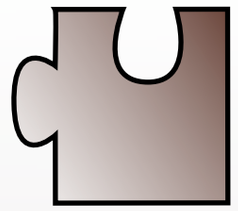
Account + RetireAccount =

```
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:

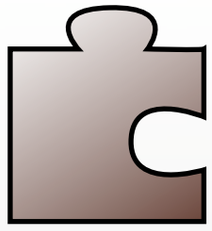
  {Account, RetireAccount, InvestAccount}

- Typically feature combinations are constrained
  - A **feature model** represents these constraints
  - Propositional formula is compact representation [Batory05]

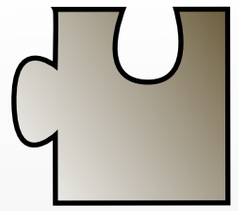
    RetireAccount ∨ InvestAccount
  - Product corresponds to truth assignment
- FMs should enforce implementation constraints
  - Safe Composition

    (RetireAccount ∨ InvestAccount) ∧

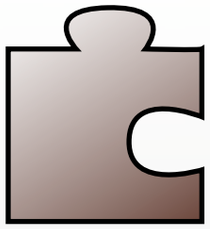
    (RetireAccount ⇒ Account) ∧ (InvestAccount ⇒ Account)



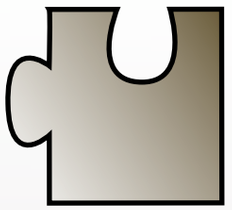
Checking Safe Composition



- Could synthesize entire product line
  - Computationally expensive:



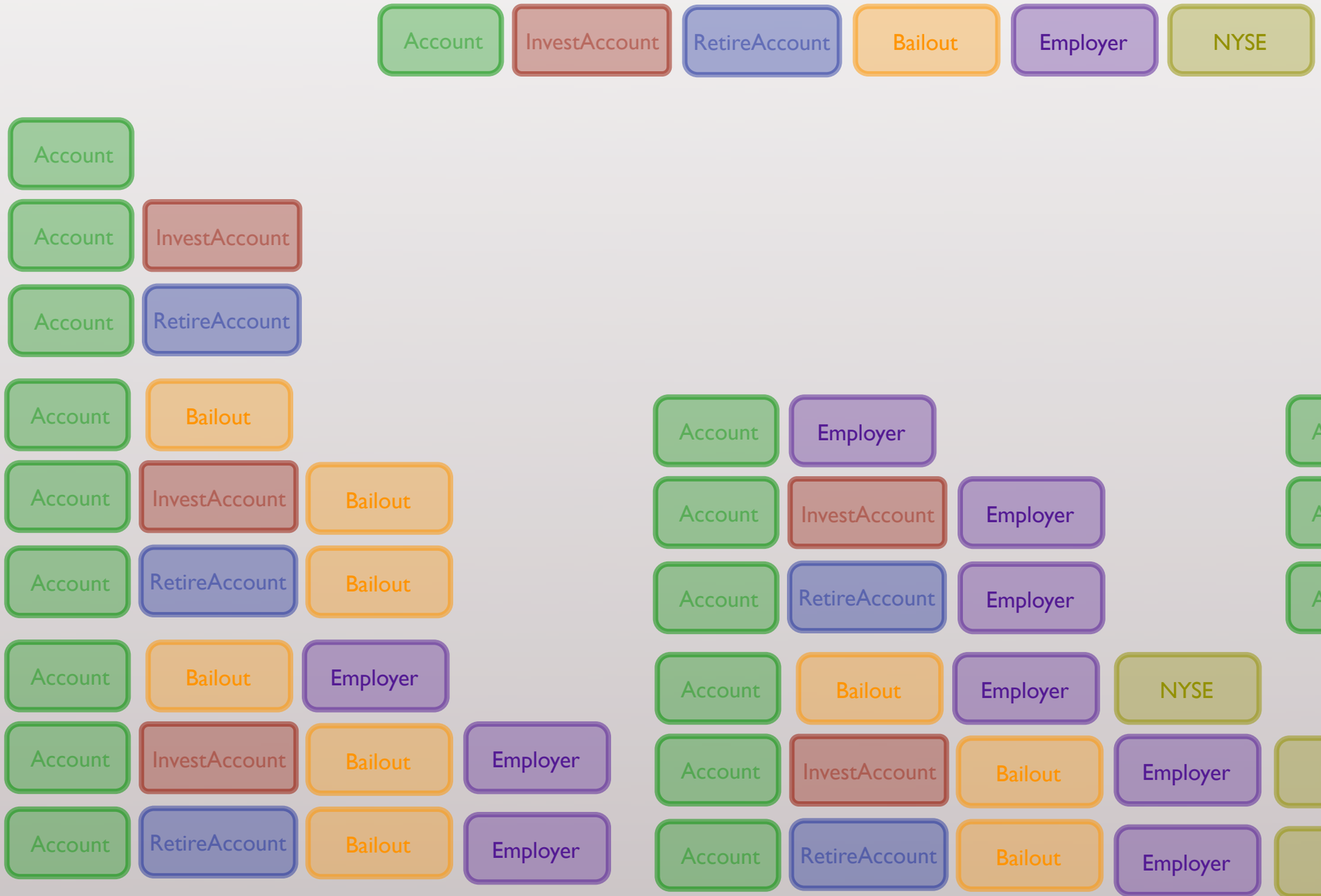
Checking Safe Composition



- Could synthesize entire product line
  - Computationally expensive:

Checking Safe Composition

- Could synthesize entire product line
  - Computationally expensive:



Difficulties

- Combinatorial nature of SPLs problematic:

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

Bailout

- 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 ::= feature F {cld; rcld}
```

Class refinement

```
  rcld ::= refines class dcl extending cl {fd; md; rmd}
```

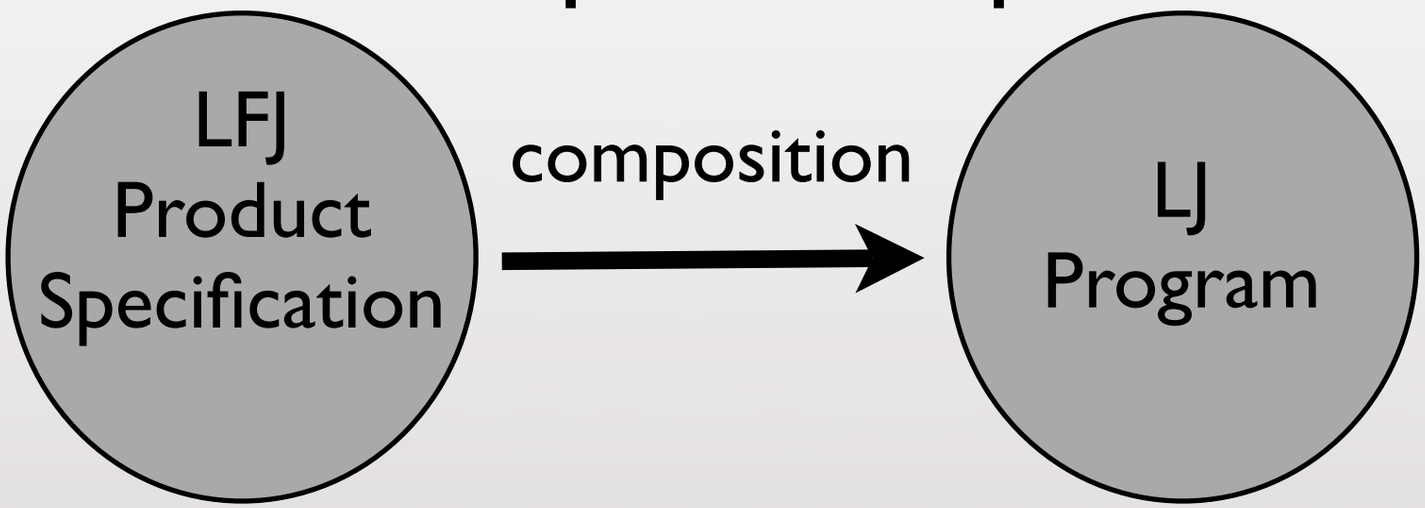
Method Refinement

```
  rmd ::= refines method ms {s; Super(); s; 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

$$\frac{\mathbf{distinct}(\overline{var_k}^k) \quad \overline{\mathbf{type}(cl_k) = \tau_k}^k \quad \mathbf{type}(cl) = \tau' \quad \Gamma = [\overline{var_k \mapsto \tau_k}^k][\mathbf{this} \mapsto \tau] \quad \Gamma(y) = \tau'' \quad \overline{\mathsf{P}, \Gamma \vdash s_\ell}^\ell \quad \mathsf{P} \vdash \tau'' \prec \tau' \quad \overline{\mathsf{P} \vdash \mathbf{defined}\ cl_k}^k}{\mathsf{P} \vdash_\tau cl\ meth\ (\overline{cl_k\ var_k}^k)\ \{\overline{s_\ell}^\ell\ \mathbf{return}\ y;\}} \quad \text{(WF-METHOD)}$$

- Program not available until composition

LJ Type System

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

Internal Checks

(WF-Method)

- Program not available until composition

LJ Type System

$$\frac{\mathbf{distinct}(\overline{var_k}^k) \quad \overline{\mathbf{type}(cl_k) = \tau_k}^k \quad \mathbf{type}(cl) = \tau' \quad \Gamma = [\overline{var_k \mapsto \tau_k}^k][\mathbf{this} \mapsto \tau] \quad \Gamma(y) = \tau'' \quad \overline{\mathsf{P}, \Gamma \vdash s_\ell}^\ell \quad \mathsf{P} \vdash \tau'' \prec \tau' \quad \overline{\mathsf{P} \vdash \mathbf{defined}\ cl_k}^k}{\mathsf{P} \vdash_\tau cl\ meth\ (\overline{cl_k\ var_k}^k)\ \{\overline{s_\ell}^\ell\ \mathbf{return}\ y;\}}\ (\text{WF-Method})$$

External Checks

- Program not available until composition

Constraint-Based Typing

- External premises become constraints

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

- Compositional Constraints
- Uniqueness Constraints
- Structural Constraints

Constraint-Based Typing

- Two typing phases
- Typing Feature Tables

$$\frac{\vdash \overline{FD_k^k} \mid \mathbf{WF}_k}{\vdash \overline{\{FD_k^k\}} \mid \bigcup_k \{\mathbf{In}_{FD_k} \Rightarrow \mathbf{WF}_k\}}$$

- Well-typed product specification

$$PS \models \bigcup_k \{\mathbf{In}_{FD_k} \Rightarrow \mathbf{WF}_k\}$$

  - Feature Constraint
  - Compositional Constraints
  - Uniqueness Constraints
  - Structural Constraints

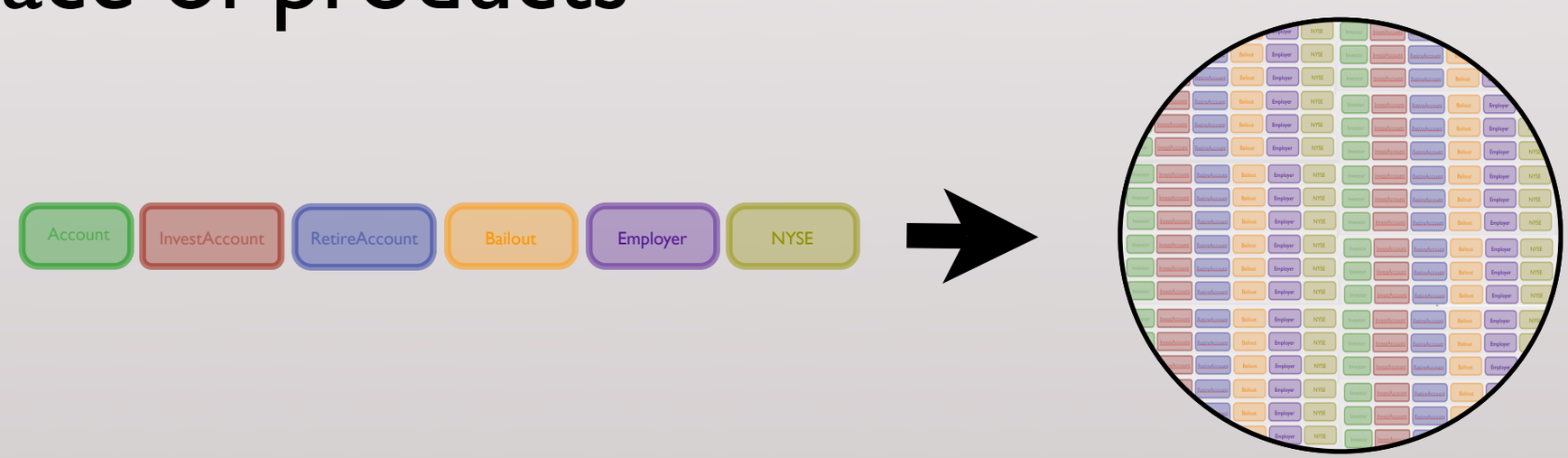
Soundness of LFJ Type System

Theorem:

$$\vdash \{FD_k\} \mid \bigcup_k \{\mathbf{In}_{FD_k} \Rightarrow \mathbf{WF}_k\}$$
$$PS \models \bigcup_k \{\mathbf{In}_{FD_k} \Rightarrow \mathbf{WF}_k\}$$

$$\vdash_{FJ} \mathbf{compose}(PS)$$

- Space of products



- First premise describes subset of type-safe products
- Second ensures product in this space

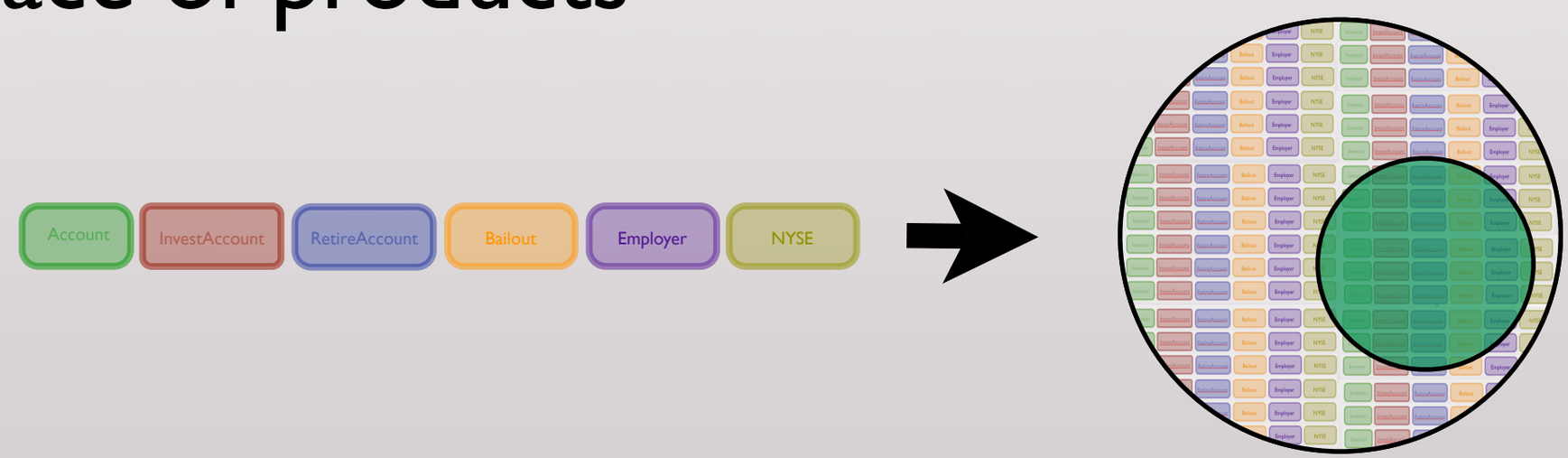
Soundness of LFJ Type System

Theorem:

$$\vdash \{FD_k\} \mid \bigcup_k \{\mathbf{In}_{FD_k} \Rightarrow \mathbf{WF}_k\}$$
$$PS \models \bigcup_k \{\mathbf{In}_{FD_k} \Rightarrow \mathbf{WF}_k\}$$

$$\vdash_{FJ} \mathbf{compose}(PS)$$

- Space of products



- First premise describes subset of type-safe products
- Second ensures product in this space

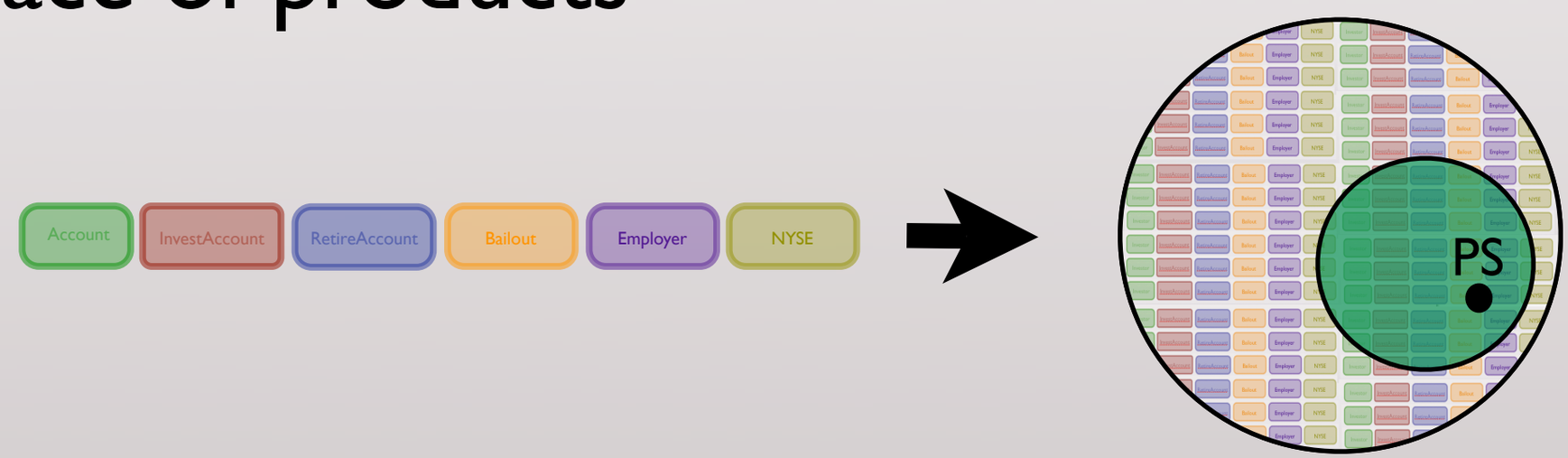
Soundness of LFJ Type System

Theorem:

$$\vdash \{FD_k\} \mid \bigcup_k \{\mathbf{In}_{FD_k} \Rightarrow \mathbf{WF}_k\}$$
$$PS \models \bigcup_k \{\mathbf{In}_{FD_k} \Rightarrow \mathbf{WF}_k\}$$

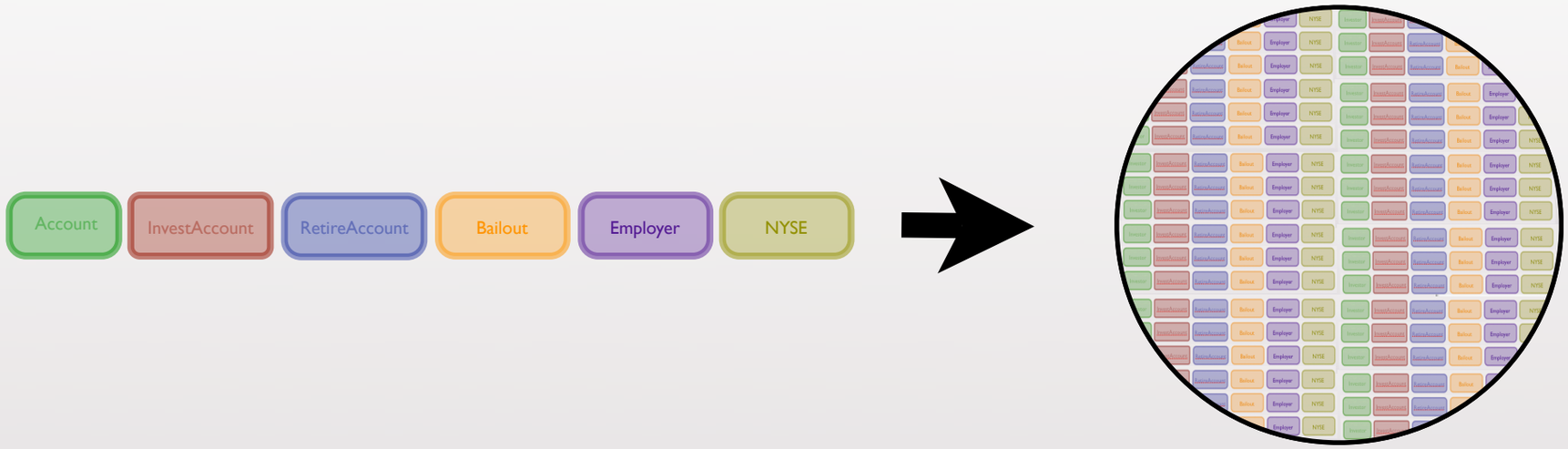
$$\vdash_{FJ} \mathbf{compose}(PS)$$

- Space of products



- First premise describes subset of type-safe products
- Second ensures product in this space

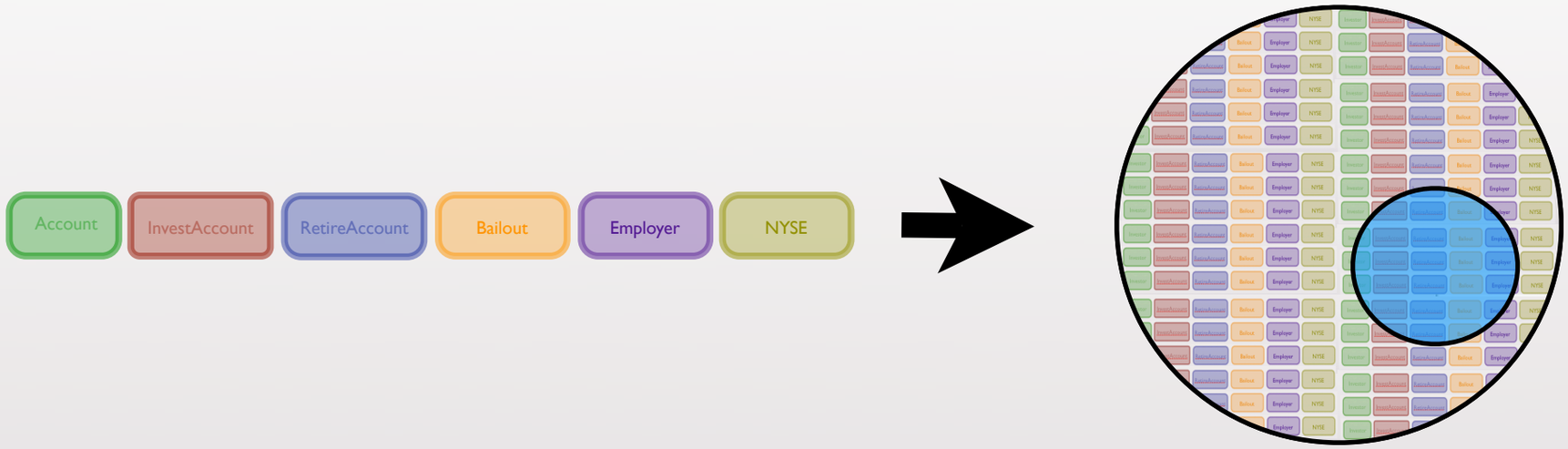
Validating Feature Models



- 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, $\mathbf{WF}_{\text{Safe}}$
  - Reduction from typing constraints
- Reduce to SAT:

$$\boxed{FM \Rightarrow \mathbf{WF}_{\text{Safe}}}$$

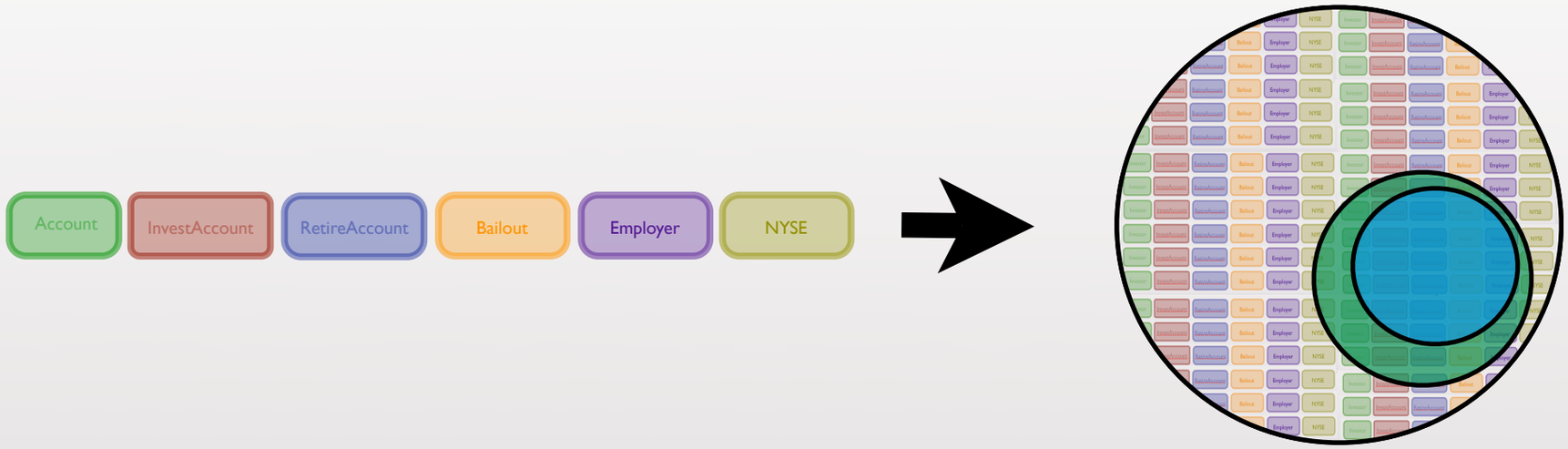
Validating Feature Models



- 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, $\mathbf{WF}_{\text{Safe}}$
  - Reduction from typing constraints
- Reduce to SAT:

$$\boxed{FM \Rightarrow \mathbf{WF}_{\text{Safe}}}$$

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, $\mathbf{WF}_{\text{Safe}}$
  - Reduction from typing constraints
- Reduce to SAT:

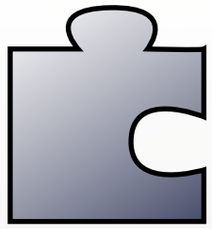
$$\boxed{FM \Rightarrow \mathbf{WF}_{\text{Safe}}}$$

Evaluation

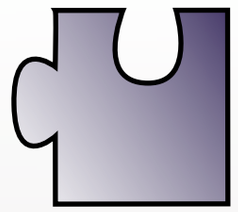
- Checking validity coNP-complete in general
- Our formulas are highly structured

Product Line	# of Features	# of Programs	Code Base Jak/Java LOC	Program Jak/ Java LOC	Typechecking Time
JPL	70	56	34K/48K	22K/35K	<30s

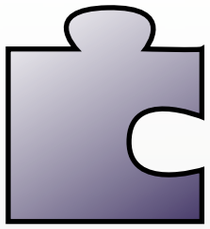
- Previous implementation of approach [Thaker07]
  - Identified errors in existing product lines
- Evidence of erroneous product



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?