Compositional Type-Checking for Delta-Oriented Programming

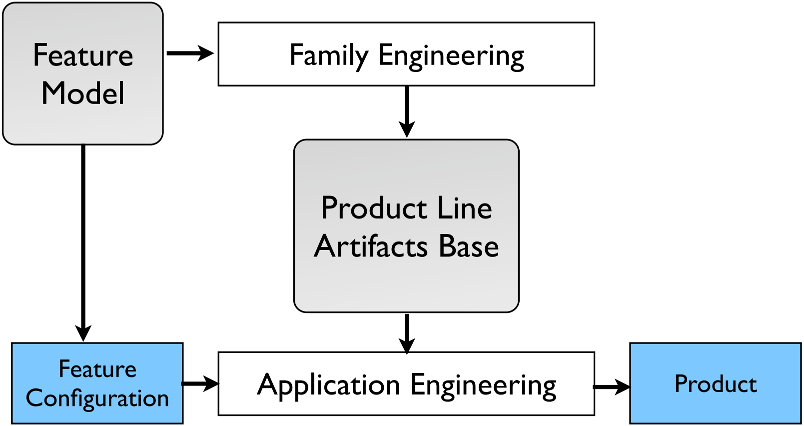
Ina Schaefer[(a)], Lorenzo Bettini[(b)] and Ferruccio Damiani[(b)]

[(a)]: TU Braunschweig, Germany
[(b)]: University of Torino, Italy

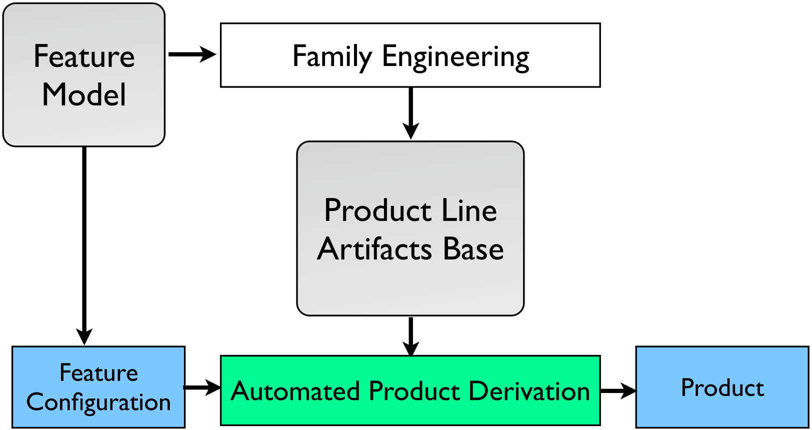
AOSD 2011

23 March 2011

Motivation


Feature Model
Family Engineering
Product Line Artifacts Base
Feature Configuration
Application Engineering
Product

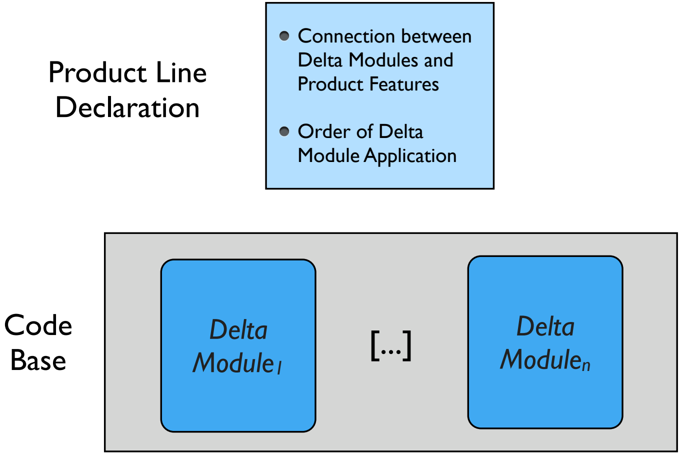
Motivation (2)



Outline

- Delta-oriented Programming (Concepts and Application)
- Compositional Type Checking for DOP
- Formalization of DOP Type Checking
- Related Work on FOP Type Checking

Delta-oriented Programming (DOP)



[Schaefer et al., SPLC 2010; Schaefer and Damiani, FOSD 2010]

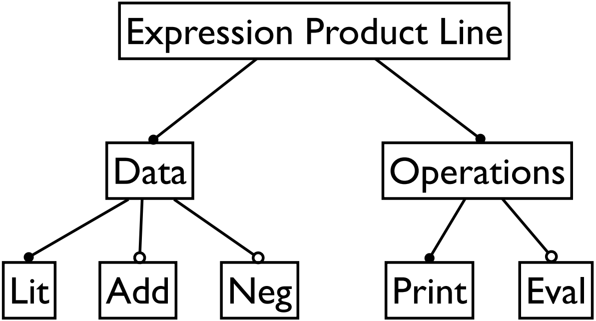
Product Generation in Delta-oriented Product Lines

Given a given feature configuration:

1. determine delta modules with valid application condition
2. apply the changes specified by delta modules
   - to the empty program
   - according to the delta module application ordering

Example: Expression Product Line (EPL)

Feature Model of EPL:



Some Delta Modules for EPL

```
delta DLit{
   adds interface Exp {
   }
   adds class Lit implements Exp {
      int value;
      Lit(int n) { value = n; }
   }
}

delta DLitPrint{
   modifies interface Exp { adds String toString();
   }
   modifies class Lit {
      adds String toString() { return value; }
   }
}

delta DLitEval{
   modifies interface Exp { adds int eval();
   }
   modifies class Lit {
      adds int eval() { return value; }
   }
}
```

Product Line Declaration for EPL

```
features Lit, Add, Neg, Print, Eval

configurations Lit & Print

deltas
   [ DLit,
     DAdd when Add,
     DNeg when Neg ]

   [ DLitPrint,
     DLitEval when Eval,
     DAddPrint when Add,
     DAddEval when (Add & Eval),
     DNegPrint when Neg,
     DNegEval when (Neg & Eval) ]

   [ DAddNegPrint when (Add & Neg) ]
```

Product for Features Lit, Add, Neg, Print

```
interface Exp { adds String toString();
}

class Lit implements Exp {
   int value;
   Lit(int n) { value = n; }
   String toString() { return value; }
}

class Add implements Exp {
   Exp expr1;
   Exp expr2
   Add(Exp a, Exp b) { expr1 = a; expr2 = b;}
   String toString() { return "(" + expr1 + " + " + expr2 + ")"; } }
}

class Neg implements Exp {
   Exp expr;
   Neg(Exp a) { expr = a; }
   String toString() { return "-" + expr; }
}
```

Software Product Line Engineering (SPLE)

Delta-oriented Programming supports

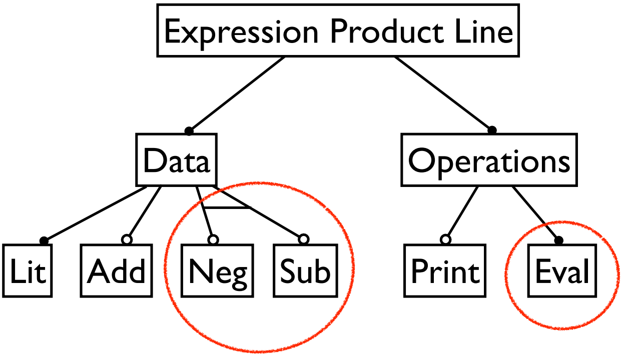
- Proactive SPLE: All products are planned in advance
- Extractive SPLE: Start from existing products
- Reactive SPLE: Evolve product line, when new features arise

Extractive Development of EPL

```
features Lit, Add, Neg, Print, Eval

configurations Lit & Print

deltas
   [ DLitNegPrint when (!Add & Neg) ] /* Existing product */

   [ DLitAddPrint when (Add | !Neg) ] /* Existing product */

   [ DNeg when (Add & Neg),
     DremAdd when (!Add & !Neg) ]     /* Feature removal */

   [ DNegPrint when (Add & Neg),
     DLitEval when Eval,
     DAddEval when (Add & Eval),
     DNegEval when (Neg & Eval) ]

   [ DAddNegPrint when (Add & Neg) ]
```

Evolution of EPL

Feature model for Evolved EPL:



Reactive Development of EPL

```
features Lit, Add, Neg, Sub, Print, Eval

configurations Lit & Eval & choose1(Neg,Sub)

deltas
   [ DLit,
     DAdd when Add,
     DNeg when Neg,
     DSub when Sub  /* new delta module */ ]

   [ DLitPrint when Print,
     DLitEval,
     DAddPrint when (Add & Print),
     DAddEval when Add,
     DNegPrint when (Neg & Print),
     DNegEval when Neg,
     DSubPrint when (Sub & Print),  /* new delta module */
     DSubEval when Sub   /* new delta module */ ]

   [ DAddNegPrint when (Add & (Neg | Sub) & Print) ]
```

Type-checking of Delta-oriented SPLs

Type-safe SPL

A SPL is type safe if all its products are well-typed programs.

Naive approach:

- Generate all the products
- Type check each product separately

Problems:

- Infeasible for large product lines
- Difficult to trace errors to delta modules

Requirements for DOP Type System

1. Check type safety without generating the products
2. Report errors in code of delta modules
3. Analyze each delta module in isolation (reusability)

Compositional Type Checking for Delta-oriented SPL

Main Idea: Define abstract product generation and analyze product abstractions for type safety.

Preliminary Step: Constraint-based type checking of programs:

Given a program (class table) $\mathtt{CT}$, infer a program abstraction $\langle \mathit{signature}(\mathtt{CT}), \mathscr{C} \rangle$ where

1. $\mathit{signature}(\mathtt{CT})$ is the *class signature table*
2. $\mathscr{C}$ a set of *class constraints*

$\langle \mathit{signature}(\mathtt{CT}), \mathscr{C} \rangle$ suffices to check that $\mathtt{CT}$ is well typed

Compositional Type Checking for Delta-oriented SPLs (2)

Step 1: Generate Abstraction of Delta Modules:

For each delta module δ infer $\langle signature(\delta), \mathscr{D}_\delta \rangle$ where

1. $signature(\delta)$ is the *delta module signature*
2. $\mathscr{D}_\delta$ a set of *delta clause-constraints*

Step 2: Generate Product Abstractions:

For each valid feature configuration $\overline{\varphi}$,

1. generate class signature table $signature(\mathtt{CT}_{\overline{\varphi}})$ from delta module signatures
2. generate class constraints $\mathscr{C}_{\overline{\varphi}}$ from delta module constraints

Step 3: Check Product Abstraction $\langle signature(\mathtt{CT}_{\overline{\varphi}}), \mathscr{C}_{\overline{\varphi}} \rangle$ to ensure that product $\mathtt{CT}_{\overline{\varphi}}$ is well typed.

Formalization

Imperative Featherweight Delta Java (IFΔJ)

An IFΔJ SPL is a 5-tuple $\mathtt{L} = (\overline{\varphi}, \Phi, \mathtt{DMT}, \Gamma, \prec)$

1. $\overline{\varphi}$ are the features of the SPL
2. $\Phi \subseteq \mathscr{P}(\overline{\varphi})$ is the set of the valid feature configurations
3. $\mathtt{DMT}$ is the delta module table (code base)
4. $\Gamma : dom(\mathtt{DMT}) \longrightarrow \Phi$ specifies for which feature configurations a delta module must be applied
5. $\prec$ is a total order on a partition of $dom(\mathtt{DMT})$

IFΔJ: Syntax of Classes and Delta Modules

Imperative Featherweight Java (IFJ)

CD	::=	class C extends C { $\overline{\text{FD}}$; $\overline{\text{MD}}$ }	classes
FD	::=	C f	fields
MD	::=	C m ($\bar{\text{C}}$ $\bar{\text{x}}$){return e; }	methods
e	::=	x | e.f | e.m($\bar{\text{e}}$) | new C() | (C)e |	expressions
		e.f = e | null | original	

Imperative Featherweight Delta Java (IFΔJ)

DMD	::=	delta δ {$\overline{\text{DC}}$}	delta modules
DC	::=	adds CD |	delta clauses
		modifies C [extending C] { $\overline{\text{DS}}$ } |	
		removes C	
DS	::=	adds FD | adds MD |	delta subclauses
		modifies MD |	
		removes a	

Constraint-based Type System for IFJ

Class constraints:

$\texttt{C}$ **with** $\mathscr{K}$	class $\texttt{C}$ has the set of method constraints $\mathscr{K}$

Method constraints:

$\texttt{m}$ **with** $\mathscr{F}$	method $\texttt{m}$ has the set of flat constraints $\mathscr{F}$

Expression constraints:

class$(\texttt{C})$	class $\texttt{C}$ must be defined
subtype(τ,η)	τ must be a subtype of η
cast$(\texttt{C},\tau)$	type τ must be castable to $\texttt{C}$
field$(\eta,\texttt{f},\alpha)$	class η must define or inherit field $\texttt{f}$ of type α
meth$(\eta,\texttt{m},\overline{\alpha}\rightarrow\beta)$	class η must define or inherit method $\texttt{m}$ of type $\overline{\alpha}\rightarrow\beta$

Constraint-based Type System for IFJ - Selected Rules

Program typing:

$$\frac{dom(\mathtt{CT}) = \{\mathtt{C}_1, \ldots, \mathtt{C}_n\} (n \geq 0) \qquad \forall i \in 1..n, \quad \vdash \mathtt{CT}(\mathtt{C}_i) : \mathtt{C}_i \ \textbf{with} \ \mathscr{K}_i}{\vdash \mathtt{CT} : \{\mathtt{C}_1 \ \textbf{with} \ \mathscr{K}_1, \ldots, \mathtt{C}_n \ \textbf{with} \ \mathscr{K}_n\}}$$

Class definition typing:

$$\frac{\forall i \in 1..q, \quad \mathtt{this} : \mathtt{C} \vdash \mathtt{MD}_i : \{\mathtt{m}_i \ \textbf{with} \ \mathscr{F}_i\}}{\vdash \mathtt{class} \ \mathtt{C} \ \mathtt{extends} \ \mathtt{D} \ \{ \ \overline{\mathtt{FD}}; \ \mathtt{MD}_1 \cdots \mathtt{MD}_q \ \} : \mathtt{C} \ \textbf{with} \ \cup_{i \in 1..q} \{\mathtt{m}_i \ \textbf{with} \ \mathscr{F}_i\}}$$

Method definition typing:

$$\frac{\mathtt{this} : \mathtt{C}, \mathtt{original} : \mathtt{B}, \bar{\mathtt{x}} : \bar{\mathtt{A}} \vdash \mathtt{e} : \tau \mid \mathscr{F}}{\mathtt{this} : \mathtt{C} \vdash \mathtt{B} \ \mathtt{m} \ (\bar{\mathtt{A}} \ \bar{\mathtt{x}})\{\mathtt{return} \ \mathtt{e}; \} : \mathtt{m} \ \textbf{with} \ (\{\textbf{subtype}(\tau, \mathtt{B})\} \cup \mathscr{F})}$$

Constraint-based Type System for IFΔJ

Delta clause-constraints:

`adds` C **with** $\mathscr{K}$	add the constraint "C **with** $\mathscr{K}$"
`removes` C	remove constraint "C **with** $\cdots$"
`modifies` C **with** $\mathscr{M}$	change the constraint "C **with** $\mathscr{K}$" into "APPLY(`modifies` C **with** $\mathscr{M}$, C **with** $\mathscr{K}$)"

Delta subclause-constraints:

`adds` m **with** $\mathscr{F}$	add the constraint "m **with** $\mathscr{F}$"
`removes` m	remove constraint "m **with** $\cdots$"
`replaces` m **with** $\mathscr{F}'$	change constraint "m **with** $\mathscr{F}$" into "m **with** $\mathscr{F}'$"
`wraps` m **with** $\mathscr{F}'$	change constraint "m **with** $\mathscr{F}$" into "m **with** $\mathscr{F} \cup \mathscr{F}'$"

Constraint-based Type System for IFΔJ - Selected Rules

Delta-module typing:

$$\frac{\forall i \in 1..n, \quad \vdash \mathtt{DC}_i : dcc_i}{\vdash \mathtt{delta}\ \delta\ \{\mathtt{DC}_1 \ldots \mathtt{DC}_n\} : \{dcc_1, \ldots, dcc_n\}}$$

Delta-clause typing:

$$\frac{\vdash \mathtt{CD} : \mathtt{C}\ \mathbf{with}\ \mathscr{K}}{\vdash \mathtt{adds}\ \mathtt{CD} : \mathtt{adds}\ \mathtt{C}\ \mathbf{with}\ \mathscr{K}}$$

$$\vdash \mathtt{removes}\ \mathtt{C} : \mathtt{removes}\ \mathtt{C}$$

$$\frac{\forall i \in 1..q, \quad \mathtt{this} : \mathtt{C} \vdash \mathtt{DS}_i : \mathscr{S}_i}{\vdash \mathtt{modifies}\ \mathtt{C}\ [\mathtt{extending}\ \mathtt{D}]\ \{\ \mathtt{DS}_1 \ldots \mathtt{DS}_q\ \} : \mathtt{modifies}\ \mathtt{C}\ \mathbf{with}\ (\cup_{i \in \{1,\ldots,q\}} \mathscr{S}_i)}$$

Constraint Application in IFΔJ Type System

The application of a set of delta clause constraints $\mathscr{D}$ to a set of class constraints $\mathscr{C}$ is the set of class constraints

$$\textsc{apply}(\mathscr{D},\mathscr{C})(\mathtt{C}) = \begin{cases} \mathscr{C}(\mathtt{C}) & \text{if } \mathtt{C} \notin \mathit{dom}(\mathscr{D}) \\ \mathtt{C}\ \textbf{with}\ \mathscr{K} & \text{if } \mathtt{C} \notin \mathit{dom}(\mathscr{C}) \\ & \quad \text{and adds } \mathtt{C}\ \textbf{with}\ \mathscr{K} \in \mathscr{D} \\ \textsc{apply}(\mathscr{D}(\mathtt{C}),\mathscr{C}(\mathtt{C})) & \text{if } \mathtt{modifies}\ \mathtt{C} \cdots \in \mathscr{D} \end{cases}$$

where $\textsc{apply}(\mathscr{D}(\mathtt{C}),\mathscr{C}(\mathtt{C}))(\mathtt{m}) =$

$$\begin{cases} \mathscr{C}(\mathtt{C})(\mathtt{m}) & \text{if } \mathtt{removes}\ \mathtt{m} \cdots \notin \mathscr{D}(\mathtt{C}) \\ & \quad \text{and } \mathtt{modifies}\ \mathtt{m} \cdots \notin \mathscr{D}(\mathtt{C}) \\ \mathtt{m}\ \textbf{with}\ \mathscr{F} & \text{if } \mathscr{D}(\mathtt{C})(\mathtt{m}) = \text{adds } \mathtt{m}\ \textbf{with}\ \mathscr{F} \\ & \quad \text{or } \mathscr{D}(\mathtt{C})(\mathtt{m}) = \mathtt{replaces}\ \mathtt{m}\ \textbf{with}\ \mathscr{F} \\ \mathtt{m}\ \textbf{with}\ \mathscr{F} \cup \mathscr{F}' & \text{if } \mathscr{D}(\mathtt{C})(\mathtt{m}) = \mathtt{wraps}\ \mathtt{m}\ \textbf{with}\ \mathscr{F}' \\ & \quad \text{and } \mathscr{C}(\mathtt{C})(\mathtt{m}) = \mathtt{m}\ \textbf{with}\ \mathscr{F} \end{cases}$$

Correctness and Completeness of IFΔJ Typing

Let $\overline{\psi} \in \Phi$ be a valid feature configuration.

(Correctness) For all $\delta \in \Gamma^{-1}(\overline{\psi})$, let $\vdash \texttt{delta}\ \delta\ \cdots : \mathscr{D}_\delta$

and let the class signature table $\texttt{CST}_{\overline{\psi}}$ for the feature configuration $\overline{\psi}$ satisfy the generated class constraints $\texttt{CST}_{\overline{\psi}} \models \mathscr{C}_{\overline{\psi}}$.

Then it holds that $\vdash \texttt{CT}_{\overline{\psi}}\ \mathrm{OK}$.

(Completeness) Let $\vdash \texttt{CT}_{\overline{\psi}}\ \mathrm{OK}$.

Then for all $\delta \in \Gamma^{-1}(\overline{\psi})$, there exists $\mathscr{D}_\delta$ with $\vdash \texttt{delta}\ \delta\ \cdots : \mathscr{D}_\delta$, such that $\texttt{CST}_{\overline{\psi}} \models \mathscr{C}_{\overline{\psi}}$.

Compositional Type Checking for FOP

[Delaware et al., FOAL 2009]

Preliminary Step: For each LFJ program infer a set of constraints: Validity of constraints ensures that program is well typed.

Step 1: For each feature module infer a set of constraints.

Step 2: For each valid feature configuration, check constraints against feature module.

Step 3' (instead of 3): From product line declaration and feature module constraints, construct a propositional formula whose satisfiability implies the type safety of the SPL.

Conclusion

Summary:

- Delta-oriented Programming
- Compositional Type Checking for DOP Product Lines

Future Work:

- Prototypical implementation and case studies [Schaefer et al., SPLC 2010]
- Add *step 3' of FOP type checking* [Delaware et al., FOAL 2009] to DOP type checking