On Type Checking Delta-Oriented Product Lines

Ferruccio Damiani       Michael Lienhardt

{ferruccio.damiani,michael.lienhardt}@di.unito.it
University of Torino (Italy)

COST Action ARVI meeting

June 5, 2016

http://hyvar-project.eu/
This presentation: mainly about the iFM 2016 paper

- A modular and tunable type system for delta-oriented programming

Structure

- Introduction
  - Software Product Line, delta-oriented programming, core calculus
- Approach
- Type System
- Future Work:
  - We need RV
Software Product Line (SPL) [Clements and Northrop, 2001]

A family of programs (called variants) generated from a common artifact base
Software Product Line (SPL) [Clements and Northrop, 2001]
A family of programs (called variants) generated from a common artifact base

Delta-Oriented Programming (DOP) [Schaefer et al., SPLC 2010]
An approach for implementing SPLs
Introduction

Structure of an SPL [Czarnecki and Eisenecke, 2000]

- **Feature Model (FM)**
  - **Features** (a feature represents an abstract description of functionality)
  - **Products** (a product is a set of features)

- **Artifact Base (AB)**
  - Set of reusable code artifacts

- **Configuration Knowledge (CK)**
  - Connects FM and AB (induces a mapping from products to variants)
Introduction

Structure of an SPL [Czarnecki and Eisenecke, 2000]

- **Feature Model (FM)**
  - Features (a feature represents an abstract description of functionality)
  - Products (a product is a set features)

- **Artifact Base (AB)**
  - Set of reusable code artifacts

- **Configuration Knowledge (CK)**
  - Connects FM and AB (induces a mapping from products to variants)

DOP: aimed at flexibility and modularity [Schaefer and Damiani, FOSD 2010]

- **The Artifact Base**
  - Base program (a Java program)
  - Deltas (sets of class operations)
    - adds, removes, modifies classes and attributes

- **The Configuration Knowledge**
  - \( \alpha \): activation conditions of deltas
  - \( < \): application order between deltas
Introduction

Structure of an SPL [Czarnecki and Eisenecke, 2000]

- **Feature Model (FM)**
  - Features (a feature represents an abstract description of functionality)
  - Products (a product is a set features)

- **Artifact Base (AB)**
  - Set of reusable code artifacts

- **Configuration Knowledge (CK)**
  - Connects FM and AB (induces a mapping from products to variants)

DOP: aimed at flexibility and modularity [Schaefer and Damiani, FOSD 2010]

- **The Artifact Base**
  - Base program (a Java program)
  - Deltas (sets of class operations )
    - adds, removes, modifies

- **The Configuration Knowledge**
  - \( \alpha \) : activation conditions of deltas
  - \( < \) : application order between deltas

Automated generation:

1. INPUT: selected features (product)
2. OUTPUT: variant generated by applying the activated deltas in the order
DOP calculus: Imperative Featherweight Java + DOP constructs
DOP calculus: Imperative Featherweight Java + DOP constructs

The IFJ calculus

\[
P ::= \overline{CD} \\
CD ::= \text{class } C \text{ extends } C \{ \overline{AD} \} \\
AD ::= FD \mid MD \\
FD ::= C \ f \\
MD ::= C \ m(C \ x) \ \{ \text{return } e; \} \\
e ::= x \mid e.f \mid e.m(\overline{e}) \mid \text{new } C() \mid (C)e \mid e.f = e \mid \text{null}
\]

- **Program**
- **Class**
- Attribute (Field or Method)
- **Field**
- **Method**
- **Expression**
DOP calculus: Imperative Featherweight Java + DOP constructs

### The IFJ calculus

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$</td>
<td>$::= \overline{CD}$</td>
<td>Program</td>
</tr>
<tr>
<td>$CD$</td>
<td>$::= \text{class } C \text{ extends } C { \overline{AD} }$</td>
<td>Class</td>
</tr>
<tr>
<td>$AD$</td>
<td>$::= FD \mid MD$</td>
<td>Attribute (Field or Method)</td>
</tr>
<tr>
<td>$FD$</td>
<td>$::= C \ f$</td>
<td>Field</td>
</tr>
<tr>
<td>$MD$</td>
<td>$::= C \ m(C \ x) \ { \text{return } e; }$</td>
<td>Method</td>
</tr>
<tr>
<td>$e$</td>
<td>$::= x \mid e.f \mid e.m(\overline{e}) \mid \text{new } C() \mid (C) e \mid e.f = e \mid \text{null}$</td>
<td>Expression</td>
</tr>
</tbody>
</table>

### The IFΔJ calculus

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L$</td>
<td>$::= FM \ CK \ AB$</td>
<td>Product Line</td>
</tr>
<tr>
<td>$AB$</td>
<td>$::= FJ \ \overline{\Delta}$</td>
<td>Artifact Base</td>
</tr>
<tr>
<td>$\Delta$</td>
<td>$::= \text{delta } d \ { \ \overline{CO} \ }$</td>
<td>Delta Module</td>
</tr>
<tr>
<td>$CO$</td>
<td>$::= \text{adds } CD \mid \text{removes } C \mid \text{modifies } C [\text{extends } C'] \ { \ \overline{AO} \ }$</td>
<td>Class Operation</td>
</tr>
<tr>
<td>$AO$</td>
<td>$::= \text{adds } AD \mid \text{modifies } MD \mid \text{removes } a$</td>
<td>Attribute Operation</td>
</tr>
</tbody>
</table>
Example

class Int extends Object { ... }

delta dExpression {

  adds class Exp extends Object {
    Int eval() { return 0; }
  }

  adds class Add extends Exp {
    Exp l, r;
    Int eval() { return l.eval() + r.eval(); }
  }

  modifies Int extends Exp {
    adds Int eval() { return this; }
  }
}

Base Program

dExpression delta

Adds Classes Exp and Add

Modifies the Super Class of Int

Adds the method eval to Int
Problem and Approach

Type Checking an SPL

- Ensure that all Variants can be Generated
- Ensure that all Variants are well-typed

Previous approaches

- [Bettini et al., Acta Inf. 1013] [Damiani and Shaefer, ISoLA 2012] implemented for ABS modeling language [Johnsen et al., 2012] by Radu Muschevici
- https://github.com/abstools/abstools/tree/master/frontend/src/abs/frontend/delta

⇒ exponential complexity

The new approach

- Reduce Type Checking to SAT (as proposed by [Thaker, et al., GPCE 2007] [Delaware et al., ESEC/FSE 2009] for FOP)
  - co-NP hard
  - Euristics in SAT solvers
- Modular: three independent analysis
- Tunable: DOP guidelines
Problem and Approach

**Type Checking an SPL**

- Ensure that all Variants can be Generated
- Ensure that all Variants are well-typed

Previous approaches [Bettini et al., Acta Inf. 1013] [Damiani and Shaefer, ISoLA 2012]


- require to iterate over every variant ⇒ exponential complexity
Problem and Approach

Type Checking an SPL

- Ensure that all Variants can be Generated
- Ensure that all Variants are well-typed

Previous approaches [Bettini et al., Acta Inf. 1013] [Damiani and Shaefer, ISoLA 2012]
- implemented for ABS modeling language [Johnsen et al., 2012] by Radu Muschevici
  https://github.com/abstools/abstools/tree/master/frontend/src/abs/frontend/delta
- require to iterate over every variant ⇒ exponential complexity

The new approach

- Reduce Type Checking to SAT (as proposed by [Thaker, et al., GPCE 2007]
  [Delaware et al., ESEC/FSE 2009] for FOP)
  - co-NP hard
  - Euristics in SAT solvers
- Modular: three independent analysis
- Tunable: DOP guidelines
Modular type system

- Partial Typing
  - Checks that *declaration* and *usage* match
    - “Standard” type checking algorithm

- Applicability
  - Checks that variants can be generated
    - SAT constraint

- Dependency
  - Checks that there are no missing dependencies in generated variants
    - SAT constraint
Problem and Approach

Tunable type system: DOP guidelines

G1: no useless operations

- **Useless operation**: declaration in the base program and adds or modifies operations in deltas that introduce code that is never present in any of the variants.
Problem and Approach

Tunable type system: DOP guidelines

**G1: no useless operations**

- **Useless operation**: declaration in the base program and adds or modifies operations in deltas that introduce code that is never present in any of the variants

**G2: type uniformity**

- **Type uniformity**: all versions of an attribute “a” accessible from a class “C” must have the same type in the artifact base

  - No alternatives
  - Simplifies the SPL (and the type system)
Problem and Approach

Tunable type system: DOP guidelines

**G1: no useless operations**
- **Useless operation**: declaration in the base program and adds or modifies operations in deltas that introduce code that is never present in any of the variants

**G2: type uniformity**
- **Type uniformity**: all versions of an attribute “a” accessible from a class “C” **must** have the same type in the artifact base
  - ⇒ No alternatives
  - ⇒ Simplifies the SPL (and the type system)

**Variant type uniformity**: all versions of an attribute “a” accessible from a class “C” **must** have the same type in all variants
Problem and Approach

Tunable type system: DOP guidelines

G1: no useless operations

- **Useless operation**: declaration in the base program and adds or modifies operations in deltas that introduce code that is never present in any of the variants

G2: type uniformity

- **Type uniformity**: all versions of an attribute “a” accessible from a class “C” must have the same type in the artifact base
  - No alternatives
  - Simplifies the SPL (and the type system)

Variant type uniformity: all versions of an attribute “a” accessible from a class “C” must have the same type in all variants

- G2 \implies variant uniformity
- \((G1 \land \text{variant uniformity}) \implies G2\)
## Problem and Approach

Tunable type system: DOP guidelines

### G1: no useless operations

- **Useless operation:** declaration in the base program and adds or modifies operations in deltas that introduce code that is never present in any of the variants

### G2: type uniformity

- **Type uniformity:** all versions of an attribute “a” accessible from a class “C” must have the same type in the artifact base
  - No alternatives
  - Simplifies the SPL (and the type system)

**Variant type uniformity:** all versions of an attribute “a” accessible from a class “C” must have the same type in all variants

- G2 ⇒ variant uniformity
- \((G1 \land \text{variant uniformity}) \Rightarrow G2\)

Other guidelines...
1. Partial Typing

Property

⇒ Ensure that Declaration and Usage Match

Example

class Int extends Object { ... }

delta dExpression {
... 
  adds class Add extends Exp {
    Exp l, r;
    Int eval() { return l.eval() + r.eval(); }
  }
}

- Must exist
- Must exist and have type () → Int
1. Partial Typing

Property

⇒ Ensure that Declaration and Usage Match

Example

class Int extends Object { ... }

delta dExpression {
  ...
  adds class Add extends Exp {
    Exp l, r;
    Int eval() { return l.eval() + r.eval(); }
  }
  ...
}

Algorithm

⇒ Similar to IFJ typing

▷ Typing Environment is all Declarations from Base Program and Deltas
2. Applicability

**Property**

⇒ Ensure that Every Variant can be Generated

**Code Generation**

1. Keep only Deltas activated by the Selected Features
2. Apply them in order

⇒ Application Constraints =

\[
\begin{pmatrix}
\text{adds } \rho & \text{ needs } \rho & \text{ absent} \\
\text{removes } \rho & \text{ needs } \rho & \text{ present} \\
\text{modifies } \rho & \text{ needs } \rho & \text{ present}
\end{pmatrix}
\]

**Algorithm**

- Generate SAT Constraints for each Operation
2. Applicability

\[
\text{appADD}(\rho) \triangleq \bigwedge_{d \neq d'} d \wedge d' \Rightarrow \bigvee d'' d''
\]

with

\[
\begin{cases}
  d, d' \in \text{add}(\rho), & d'' \in \text{remove}(\rho) \\
  d <_L d'' <_L d'
\end{cases}
\]

If two deltas add $\rho$

there must exist a delta in between that removes it
2. Applicability

There must be a delta adding before $\rho$

\[
\text{appRM}(\rho) \triangleq \bigwedge d \ d \Rightarrow (\bigvee d_1 \ d_1 \land \bigwedge d' (d' \Rightarrow \bigvee d_2 \ d_2)) \quad \text{with} \quad \begin{cases} 
    d, d' \in \text{remove}(\rho) \\
    d_1, d_2 \in \text{add}(\rho) \\
    d_1 <_L d <_L d_2 <_L d'
\end{cases}
\]

If there is a delta removing $\rho$

and if there is another delta removing $\rho$

there must be one in between that adds it
2. Applicability

There must exist a delta before adding $\rho$

If there is a delta modifying $\rho$

with no removes $\rho$ in between
2. Applicability

**Applicability: Global Constraint**

$L$ is *applicability-consistent* iff $\Phi$ is **valid**

$$\Phi \triangleq L.FMandCK \Rightarrow \bigwedge_{\rho \in \text{add}(L)} \text{appADD}(\rho) \land \text{appRM}(\rho) \land \text{appMOD}(\rho)$$

All the products of $L$
3. Dependency

**Property**

⇒ Ensure that Variant’s code have no **Missing Dependencies**

**Example**

```java
class Int extends Object {
    ...
}

delta dExpression {
    ...
    adds class Add extends Exp {
        Exp l, r;
        Int eval() {
            return l.eval() + r.eval();
        }
    }
    ...
}
```

Must be present when `dExpression` is activated
Type System (enforcing G2)

3. Dependency

Property

⇒ Ensure that Variant’s code have no Missing Dependencies

Example

```java
class Int extends Object { ... }
delta dExpression {
    ...
    adds class Add extends Exp {
        Exp l, r;
        Int eval() { return l.eval() + r.eval(); }
    }
    ...
}
```

Algorithm

- Generate SAT Constraint for each Dependency
- Induction over all the declarations

Must be present when dExpression is activated
3. Dependency

**Induction Rules:** for Expressions

\[ \Gamma \vdash e : T \mid \Phi \]

Typing Environment:
for Dependencies on C.a

Generated Constraint

Analyzed Expression

Type of e
3. Dependency

Examples:

\[
\frac{\Gamma \vdash e : C \mid \Phi \quad \text{type}(C.f) = C'}{\Gamma \vdash e.f : C' \mid \phi \land \text{decl}(C.f) }
\]

\[
\frac{\Gamma \vdash e : C \mid \Phi \quad \text{type}(C.m) = (C_1, \ldots, C_n) \to C'}{\Phi' = \text{sub}(T_i, C_i)}
\]

\[
\frac{\Gamma \vdash e : T_i \mid \Phi_i \quad \Phi'_i = \text{sub}(T_i, C_i) \quad \phi \land \text{decl}(C.m)}{\Gamma \vdash \land_{i}(\Phi_i \land \Phi'_i) \land \phi \land \text{decl}(C.m) }
\]

*Constraint encoding when C.f is present*

*Arguments must be a subtype of Formal Parameter*

*C.m must be present*
3. Dependency

Induction Rules: for Declaration

\[ \Phi \vdash D : \Phi \]

Declaration Environment: Contains

- \(d\): Analyzed Delta
- \(C\): analyzed Class

Analyzed Declaration

Generated Constraint
3. Dependency

Examples:

\[ (D:FIELD) \]

\[ d, C \vdash C' f : \neg rm(d, C.f) \Rightarrow decl(C') \]

If the Field is not removed

C' must be present

If the Field is not removed
Type System (enforcing G2)

3. Dependency

Examples:

If the Field is not removed

\( \neg \text{rm}(d, C.f) \Rightarrow \text{decl}(C') \)

\( C' \) must be present

If the Class is not removed

then its inner dependencies must hold

\( \bigwedge_i \Phi_i \wedge (\neg \text{modifyEC}(d, C) \Rightarrow \text{decl}(C') \wedge \neg \text{sub}(C', C)) \)

and if its super is not changed

\( C' \) must exist and is not a subtype of \( C \)

\( d, C \vdash C' \quad \text{and} \quad C' \quad \text{extends} \quad C' \quad \{AD_1 \ldots FD_n\} \)

\( d, C \vdash \text{class} \ C \quad \text{extends} \quad C' \quad \{AD_1 \ldots FD_n\} \)

\( \neg \text{rm}(d, C) \Rightarrow \bigwedge_i \Phi_i \wedge (\neg \text{modifyEC}(d, C) \Rightarrow \text{decl}(C') \wedge \neg \text{sub}(C', C)) \)
Type System (enforcing G2)

Theorem

Partially Typed + Applicability + Dependency
⇔
SPL type checks

Prototype Implementation at https://github.com/gzoumix/IFDJTS
Theorem

Partially Typed + Applicability + Dependency
⇔
SPL type checks

 Prototype Implementation at
https://github.com/gzoumix/IFDJTS
Future Work

- A better implementation.
  E.g., for ABS, for DOP on full Java [Koscielny et al., PPPJ 2014]
- Case studies
- Delta-oriented Multi SPLs
Future Work

Starting from:

- Delta-oriented dynamic SPLs [Damiani et al, GPCE 2012]
  - Extends DOP with a dynamic reconfiguration graph (runtime reconfiguration of code and heap)
  - Ensures type safety

- Formal verification of delta-oriented SPLs (ongoing work [Hähnle and Schaefer, ISoLA 2012] [Damiani et al, FMSPLE 2012 @ SPLC],...)

use RV techniques to

- Monitor the behavior of a variant
- Decide how and when trigger a reconfiguration
Thanks