A Formal Foundation for Dynamic Delta-Oriented Software Product Lines[1]

Ferruccio Damiani[(a)], Luca Padovani[(a)] and Ina Schaefer[(b)]

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

CINA - WP1 Meeting

July 23-24, 2014

Genova, Italy

[1] Talk based on the paper presented at GPCE 2012 — EAPLS GPCE/SLE 2012 Best Paper Award

Motivation

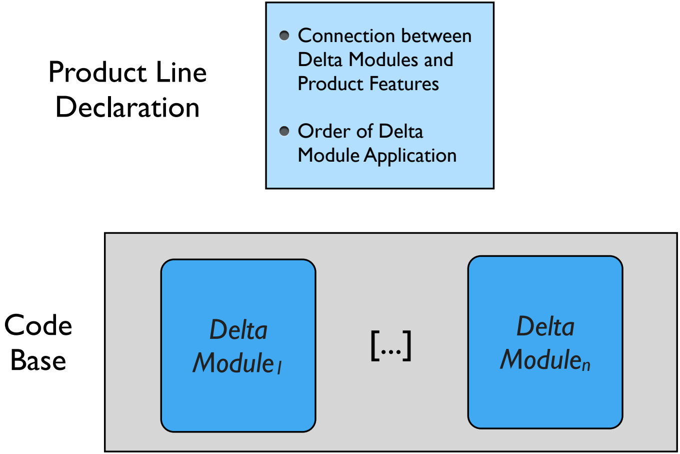
- DOP has been used for modelling variability at compile time
- This paper extends it for
  1. modelling variability at run time
  2. providing a framework for unanticipated software evolution

Outline

- Delta-oriented Programming
- Dynamic DOP
- Evolving Dynamic DOP SPLs
- Dynamic DOP (cont.)
- Formalization
- Related/Future Work

Delta-oriented Programming (DOP): concepts

A DOP SPL consists of:



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 a suitable delta module application ordering

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 a suitable delta module application ordering

Type-safe SPL

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

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 a suitable delta module application ordering

Type-safe SPL

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

- A type system that ensures type safety of DOP SPLs of IMPERATIVE FEATHERWEIGHT JAVA programs [Bettini et al., *Acta Inf.*, 2013]

Dynamic DOP: concepts

A Dynamic DOP SPL consists of:

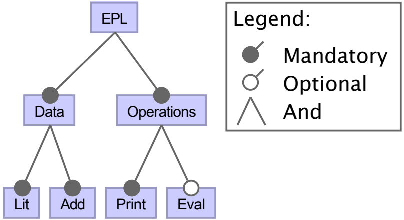
1. Product-line code base
2. Product-line declaration

3. Product-line dynamic reconfiguration graph

Example: (Dynamic) Expression Product Line

```
Exp ::= Lit | Add
Lit   ::= <non-negative integers>
Add ::= Exp "+" Exp
```

Feature Model of (Dynamic) EPL:



There are 2 valid feature configurations:

LAP	=	Lit, Add, Print
LAPE	=	Lit, Add, Print, Eval

1. Product-Line Code Base for (Dynamic) EPL

```
delta DLitAddPrint{
   adds class Exp extends Object { ... }
   adds class Lit extends Exp { ... }
   adds class Add extends Exp { ... }
   adds class Printer { ... }
   adds class Main { ... }
   adds class ExpScanner ...
}

delta DLitEval {
   modifies Exp { ... }
   modifies Lit { ... }
}

delta DAddEval {
   modifies Add { ... }
}

delta DPrinterEval {
   modifies Printer { ... }
}
```

Product-Line Code Base for EPL (1/2)

```
delta DLitAddPrint{
   adds class Exp extends Object { // Exp is only used as a type
       int counter; // number of operations invoked on the expression
       String toString() { return ""; }
   }
   adds class Lit extends Exp {
      int value; // value >= 0
      Lit setLit(int n) { value = n; return this; }
      String toString() { counter++; return value + ""; }
   }
   adds class Add extends Exp {
      Exp expr1;
      Exp expr2;
      Add setAdd(Exp a, Exp b) { expr1 = a; expr2 = b; return this; }
      String toString() { counter++; return expr1.toString() + "+" + expr2.toString(); }
   }
   adds class Printer {  String compute(Exp a) { return a.toString(); }
   }
   adds class Main {
      void main() {
         Scanner in = new ExpScanner(System.in);
         Printer pr = new Printer();
         while (true) {
            Exp expr = in.nextExp();
            System.out.println(pr.compute(expr));
         }
      }
   }
   adds class ExpScanner ... // parses data of Exp type
}
```

Product-Line Code Base for EPL (2/2)

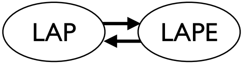
```
delta DLitEval {
   modifies Exp {
      adds int eval() { return 0; }
   }
   modifies Lit {
      adds int eval() { counter++; return value; }
   }
}

delta DAddEval {
   modifies Add {
      adds int eval() { counter++; return expr1.eval() + expr2.eval(); }
   }
}

delta DPrinterEval {
   modifies Printer {
      modifies String compute(Exp a) { return original(a) + "=" + a.eval(); }
   }
}
```

2. Product-Line Declaration for (Dynamic) EPL

```
features
   Lit, Add, Print, Eval
configurations
   Lit & Add & Print
deltas
   { DLitAddPrint }
   { DLitEval when Eval,  DAddEval when Eval,  DPrinterEval when Eval }
```

3. Dynamic Reconfiguration Graph for Dynamic EPL

Graphical representation:



Textual representation:

```
nodes
   LAP = Lit, Add, Print;
   LAPE = Lit, Add, Print, Eval;
edges
   LAP => LAPE { }
   LAPE => LAP { }
```

Runtime reconfiguration: changing the configuration of the running product

- A class C is affected by a reconfiguration if
  - either C is recoded (the code of C or of a superclass of C is modified)
  - or C is reallocated (the instances of C are changed)

- A reconfiguration is enabled if
  - classes of receivers of methods on the call stack are not affected by the reconfiguration.

Product LAP of the Dynamic EPL

```
class Exp extends Object { // Exp is only used as a type
    int counter; // number of operations invoked on the expression
    String toString() { return ""; }

}
class Lit extends Exp {
   int value; // value >= 0
   Lit setLit(int n) { value = n; return this; }
   String toString() { counter++; return value + ""; }

}
class Add extends Exp {
   Exp expr1;
   Exp expr2;
   Add setAdd(Exp a, Exp b) { expr1 = a; expr2 = b; return this; }
   String toString() { counter++; return expr1.toString() + "+" + expr2.toString(); }

}
class Printer {

   String compute(Exp a) { return a.toString(); }
}
class Main {
   void main() {
      Scanner in = new ExpScanner(System.in);
      Printer pr = new Printer();
      while (true) {
         Exp expr = in.nextExp();
         System.out.println(pr.compute(expr));
      }
}
class ExpScanner ... // parses data of Exp type
```

Product LAPE of the Dynamic EPL

```
class Exp extends Object { // Exp is only used as a type
    int counter; // number of operations invoked on the expression
    String toString() { return ""; }
    int eval() { return 0; } // ***ADDED***
}
class Lit extends Exp {
   int value; // value >= 0
   Lit setLit(int n) { value = n; return this; }
   String toString() { counter++; return value + ""; }
   int eval() { counter++; return value; } // ***ADDED***
}
class Add extends Exp {
   Exp expr1;
   Exp expr2;
   Add setAdd(Exp a, Exp b) { expr1 = a; expr2 = b; return this; }
   String toString() { counter++; return expr1.toString() + "+" + expr2.toString(); }
   int eval() { counter++; return expr1.eval() + expr2.eval(); } // ***ADDED***
}
class Printer {
   String compute$DPrinterEval(Exp a) { return a.toString(); } // ***ADDED***
   String compute(Exp a) { return compute$DPrinterEval(a) + "=" + a.eval(); } // ***MODIFIED***
}
class Main { // ***UNCHANGED***
   void main() {
      Scanner in = new ExpScanner(System.in);
      Printer pr = new Printer();
      while (true) {
         Exp expr = in.nextExp();
         System.out.println(pr.compute(expr));
      }
}
class ExpScanner ... // parses data of Exp type // ***CHANGED***
```

Unanticipated Evolution

WHAT:

- New products are added
- Existing products (different from the currently running product) are modified or removed

Unanticipated Evolution

WHAT:

- New products are added
- Existing products (different from the currently running product) are modified or removed

HOW:

Changing the DOP product-line code base, declaration, dynamic reconfiguration graph by preserving the currently running product.

Unanticipated Evolution

WHAT:

- New products are added
- Existing products (different from the currently running product) are modified or removed

HOW:

Changing the DOP product-line code base, declaration, dynamic reconfiguration graph by preserving the currently running product.

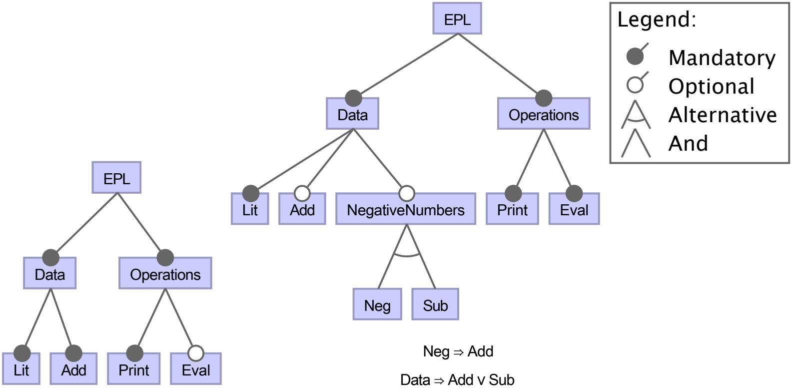
Preserved product

A product is preserved by an evolution if in both the SPLs:

- the feature configuration of the product is valid, and
- the feature configuration selects the same delta modules (in the same order).[a]

[a] This implies that code of the selected delta modules is unchanged.

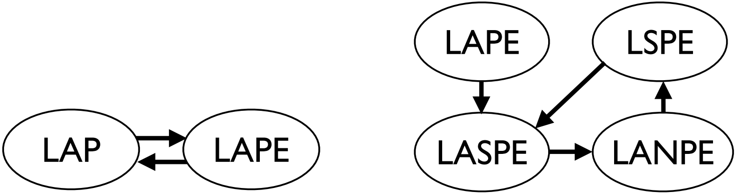
Example: Evolving the Feature Model of the Dynamic EPL



There are 4 valid feature configurations:

~~**LAP**~~	~~$\neq$~~	~~Lit, Add, Print~~
LAPE	=	Lit, Add, Print, Eval
LASPE	=	Lit, Add, Sub, Print, Eval
LANPE	=	Lit, Add, Neg, Print, Eval
LSPE	=	Lit, Sub, Print, Eval

Example: Evolving the Dynamic Reconfiguration Graph of the Dynamic EPL



There are four (valid) feature configurations:

~~**LAP**~~	~~=~~	~~Lit, Add, Print~~
LAPE	=	Lit, Add, Print, Eval
LASPE	=	Lit, Add, Sub, Print, Eval
LANPE	=	Lit, Add, Neg, Print, Eval
LSPE	=	Lit, Sub, Print, Eval

1. Product-Line Code Base of the evolved Dynamic EPL (added delta modules)

```
delta DNeg  {
   adds class Neg extends Exp {
      Exp expr;
      Neg setNeg(Exp a) { expr = a; return this; }
      String toString() { counter++; return "(" +  "-" +  expr.toString() + ")"; }
      int eval() { counter++; return (-1) * expr.eval(); }
   }
   modifies class ExpScanner ... // parses Neg expressions
}

delta DSub  {
   adds class Sub extends Exp {
      Exp expr1;
      Exp expr2;
      Add setSub(Exp a, Exp b) { expr1 = a; expr2 = b; return this; }
      String toString() { counter++; return  expr1.toString() + "-" + expr2.toString(); }
      int eval() { counter++; return expr1.eval() - expr2.eval(); }
   }
   modifies class ExpScanner ... // parses Sub expressions
}

delta DremAdd {
   removes Add
   modifies class ExpScanner ... // removes code for parsing Add expressions
}
```

2. Product-Line Declaration of the evolved Dynamic EPL

```
features
    Lit, Add, Neg, Sub, Print, Eval
configurations
    Lit & Print & Eval & (Add|Sub) & (Neg->Add) & !(Neg&Sub)
deltas
   { DLitAddPrint }
   { DLitEval,   DAddEval when Add,   DPrinterEval }
   { DSub when Sub }
   { DNeg when Neg }
   { DremAdd when !Add }
```

3. Dynamic Reconfiguration Graph of the evolved Dynamic EPL

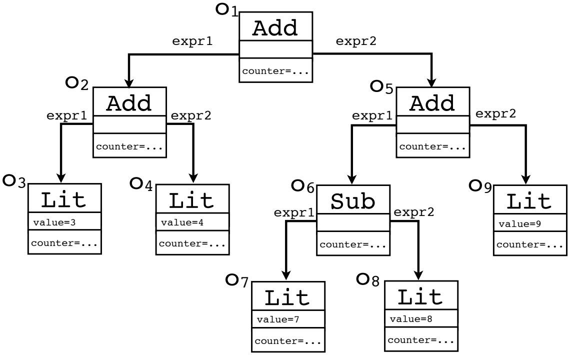
```
nodes
   LAPE = Lit, Add, Print, Eval;
   LASPE = Lit, Add, Sub, Print, Eval;
   LANPE = Lit, Add, Neg, Print, Eval;
   LSPE = Lit, Sub, Print, Eval;
edges
   LAPE => LASPE { }
   LASPE => LANPE { // e.g.:  (7-8)  becomes  (7+(-8))
         Sub->Add { pre:
                          Exp y1 = this.expr1; Exp y2 = this.expr2;
                    post:
                          Exp z1 = y1; Exp z2 = y2; Exp z3 = new Neg(z2);
                          this.counter = 0; this.expr1 = z1;
                          this.expr2 = z3; }
   }
   LANPE => LSPE {
         Neg->Sub { pre:
                          Exp y = this.expr;
                    post:
                          Exp z1 = new Lit(0); Exp z2 = y;
                          this.counter = 0; this.expr1 = z1; this.expr2 = z2; }
         Add->Sub { pre:
                          Exp y1 = this.expr1; Exp y2 = this.expr2;
                    post:
                          Exp z1 = y1; Exp z2 = y2;
                          Exp z3 = new Lit(0); Exp z4 = new Sub(z4,z3);
                          this.counter = 0; this.expr1 = z1;
                          this.expr2 = z4; }
   }
   LSPE => LASPE { }
```

Example: reconfiguration LASPE => LANPE in the evolved Dynamic EPL

$(3+4)+((7-8)+9)$ becomes $(3+4)+((7+(-8))+9)$

Affects classes

- ExpScanner (without reallocating its object)
- Sub (by dropping it and reallocating its object to class Add)



Runtime reconfiguration: changing the configuration of the running product

- A class C is affected by a reconfiguration if
  - either C is recoded (the code of C or of a superclass of C is modified)
  - or C is reallocated (the instances of C are changed)

- A reconfiguration is enabled if
  - classes of receivers of methods on the call stack are not affected by the reconfiguration.

Lazy Heap Update

Each object is reallocated when the running product accesses it.

- The queue of the ($n \geq 0$) pending reconfigurations is maintained.
- The heap is partitioned in $n+1$ regions.
  - 0-st region contains non-reallocated objects,
  - 1-nd region contains objects reallocated by the 1-st pending reconfiguration,

    $\vdots$

  - i-th region ($0 \leq i \leq n$) contains objects updated by pending reconfigurations $1, \ldots, i-1$ (the n-th region contains fully reallocated objects).

Imperative Featherweight Dynamic Delta Java (IFDΔJ)

STARTING POINT: Imperative Featherweight Delta Java (IFΔJ): a calculus for SPLs of IMPERATIVE FEATHERWEIGHT JAVA programs [Bettini et al., *Acta. Inf.*, 2013]

- Product generation
- Type system ensuring type safety

THIS PAPER: Extends IFΔJ to dynamic DOP:

1. Dynamic reconfiguration graph
2. Typing rules for the dynamic reconfiguration graph
3. Operational semantics with lazy heap (small step reduction)
4. Type soundness (subject reduction + progress)

IFΔJ: Syntax of Classes and Delta Modules

Imperative Featherweight Java (IFJ)

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

IFΔJ: Syntax of Classes and Delta Modules

Imperative Featherweight Java (IFJ)

$$
\begin{array}{llll}
\mathtt{CD} & ::= & \mathtt{class}\ \mathtt{C}\ \mathtt{extends}\ \mathtt{C}\ \{\ \overline{\mathtt{FD}};\ \overline{\mathtt{MD}}\ \} & \text{classes} \\
\mathtt{FD} & ::= & \mathtt{C}\ \mathtt{f} & \text{fields} \\
\mathtt{MD} & ::= & \mathtt{C}\ \mathtt{m}\ (\bar{\mathtt{C}}\ \bar{\mathtt{x}})\{\mathtt{return}\ \mathtt{e};\} & \text{methods} \\
\mathtt{e} & ::= & \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.m}(\bar{\mathtt{e}}) \mid \mathtt{new}\ \mathtt{C}() \mid (\mathtt{C})\mathtt{e} \mid & \text{expressions} \\
 & & \mathtt{e.f} = \mathtt{e} \mid \mathtt{null} \mid \color{red}{\mathtt{original}} &
\end{array}
$$

Imperative Featherweight Delta Java (IFΔJ)

$$
\begin{array}{llll}
\mathtt{DM} & ::= & \mathtt{delta}\ \delta\ \{\overline{\mathtt{CO}}\} & \text{delta modules} \\
\mathtt{CO} & ::= & \mathtt{adds}\ \mathtt{CD} \mid & \text{class operations} \\
 & & \mathtt{modifies}\ \mathtt{C}\ [\mathtt{extending}\ \mathtt{C}]\ \{\ \overline{\mathtt{AO}}\ \} \mid & \\
 & & \mathtt{removes}\ \mathtt{C} & \\
\mathtt{AO} & ::= & \mathtt{adds}\ \mathtt{FD} \mid \mathtt{adds}\ \mathtt{MD} \mid & \text{attribute operations} \\
 & & \mathtt{modifies}\ \mathtt{MD} \mid & \\
 & & \mathtt{removes}\ \mathtt{a} &
\end{array}
$$

IFDΔJ: Dynamic Reconfiguration Graph

Syntax

$\mathtt{R}$	$::=$	$\overline{\psi} \Rightarrow \overline{\psi}' \ \{\overline{\mathtt{OR}}\}$	reconfiguration declarations
$\mathtt{OR}$	$::=$	$\mathtt{C} \rightarrow \mathtt{C}' \ \{\mathtt{pre:}\ \overline{\mathtt{A\,y} = \mathtt{p};}$ $\mathtt{post:}\ \overline{\mathtt{B\,z} = \mathtt{q};}\ \overline{\mathtt{this.f} = \mathtt{z};}\}$	object reallocations
$\mathtt{p}$	$::=$	$\mathtt{this} \mid \mathtt{p.f}$	pre-reconfiguration expressions
$\mathtt{q}$	$::=$	$\mathtt{y} \mid \mathtt{null} \mid \mathtt{new}\ \mathtt{C}(\overline{\mathtt{z}})$	post-reconfiguration expressions

IFDΔJ: Dynamic Reconfiguration Graph

Syntax

R	::=	$\overline{\psi} \Rightarrow \overline{\psi}' \; \{\overline{\mathtt{OR}}\}$	reconfiguration declarations
OR	::=	$\mathtt{C} \rightarrow \mathtt{C}' \; \{\mathtt{pre:} \; \overline{\mathtt{A\,y} = \mathtt{p};}$ $\mathtt{post:} \; \overline{\mathtt{B\,z} = \mathtt{q};} \; \overline{\mathtt{this.f} = \mathtt{z};}\}$	object reallocations
p	::=	$\mathtt{this} \mid \mathtt{p.f}$	pre-reconfiguration expressions
q	::=	$\mathtt{y} \mid \mathtt{null} \mid \mathtt{new}\,\mathtt{C}(\overline{\mathtt{z}})$	post-reconfiguration expressions

Typing

1. If (in $\overline{\psi}$) C is a subclass of $\mathtt{C}_0$ and $\mathtt{C}_0$ is not removed by R, then (in $\overline{\psi}'$) R(C) is a subclass of $\mathtt{C}_0$.

2. If y of type A is assigned to z of type B, then the objects of every subclass D of A (in $\overline{\psi}$) are reallocated to a class D′ that is a subclass of B (in $\overline{\psi}'$).

IFDΔJ semantics: address, value, object, stack and heap

Addresses, ranged over by the metavariable ι, are the elements of the denumerable set $\mathbf{I}$.

Values, ranged over by the metavariable $\mathtt{v}$, are either addresses or $\mathtt{null}$.

Objects are denoted by $\langle \mathtt{C}, \overline{\mathtt{f}} = \overline{\mathtt{v}} \rangle$, where $\mathtt{C}$ is the class of the object, $\overline{\mathtt{f}}$ are the names of the fields and $\overline{\mathtt{v}}$ are the values of the fields.

A *stack* $\overline{\iota}$ is a possibly empty sequence of addresses (possibly containing duplicates). The empty stack is denoted by $\bullet$.

A *heap* $\mathscr{H}$ is a mapping from *addresses* to *objects*. The empty heap is denoted by $\emptyset$.

IFDΔJ semantics: lazy heap

Lazy heaps are defined as follows $\boxed{\mathscr{L} \;::=\; \mathscr{H} \quad | \quad \mathscr{H}:\mathtt{R}(\mathscr{L})}$.
That is, a lazy heap is either a heap $\mathscr{H}$ or a partially reconfigured heap of the form $\mathscr{H}_n:\mathtt{R}_n(\mathscr{H}_{n-1}:\mathtt{R}_{n-1}(\cdots\mathscr{H}_1:\mathtt{R}_1(\mathscr{H}_0)\cdots))$, for some $n \geq 1$, where

- $\mathscr{H}_n$ is the part of heap that has been reconfigured by $\mathtt{R}_n,\ldots,\mathtt{R}_1$ and that may have been subsequently modified by the execution of non-reconfiguration operations.
- each $\mathscr{H}_i$ $(1 \leq i \leq n-1)$ is the part of heap that has been reconfigured by $\mathtt{R}_i,\ldots,\mathtt{R}_1$ and that may have been subsequently modified by the execution of non-reconfiguration operations before the invocation of $\mathtt{R}_{i+1}$.
- $\mathscr{H}_0$ is the heap before the invocation of $\mathtt{R}_1$.

IFDΔJ: semantics: reduction rules

$$\text{(R-EVAL)}\quad \frac{\overline{\psi}, \mathscr{L}, \overline{\imath}, e \longrightarrow \overline{\psi}, \mathscr{L}', \overline{\imath}', e'}{\overline{\psi}, \mathscr{L}, \overline{\imath}, e \Longrightarrow \overline{\psi}, \mathscr{L}', \overline{\imath}', e'}$$

$$\text{(R-RECONF)}\quad \frac{\mathtt{R} = \overline{\psi} \Rightarrow \overline{\psi}'\{\cdots\} \qquad \textbf{Enabled}(\mathtt{R}, \mathscr{L}, \overline{\imath}, e)}{\overline{\psi}, \mathscr{L}, \overline{\imath}, e \Longrightarrow \overline{\psi}', \emptyset : \mathtt{R}(\mathscr{L}), \overline{\imath}, e}$$

IFDΔJ: semantics: computation rules

(C-New)

$$\frac{\iota \text{ fresh} \qquad \mathit{fields}_{\overline{\psi}}(\mathtt{C}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}}{\_, \mathscr{L}, \overline{\iota}, \mathtt{new\ C()} \longrightarrow \_, \mathscr{L} \cup \{\iota \mapsto \langle \mathtt{C}, \overline{\mathtt{f}} = \overline{\mathtt{null}} \rangle\}, \overline{\iota}, \iota}$$

(C-Field)

$$\frac{\mathit{olookup}(\iota, \mathscr{L}) = \langle \mathtt{C}, \overline{\mathtt{f}} = \overline{\mathtt{v}} \rangle, \mathscr{L}'}{\_, \mathscr{L}, \overline{\iota}, \iota.\mathtt{f}_i \longrightarrow \_, \mathscr{L}', \overline{\iota}, \mathtt{v}_i}$$

(C-Assign)

$$\frac{\mathit{olookup}(\iota, \mathscr{L}) = \langle \mathtt{C}, \overline{\mathtt{f}} = \overline{\mathtt{v}} \rangle, \mathscr{L}'}{\_, \mathscr{L}, \overline{\iota}, \iota.\mathtt{f}_i = \mathtt{v} \longrightarrow \_, \mathscr{L}'[\iota \mapsto \langle \mathtt{C}, \ldots, \mathtt{f}_i = \mathtt{v}, \ldots \rangle], \overline{\iota}, \mathtt{v}}$$

(C-Invk)

$$\frac{\begin{array}{c}\mathit{olookup}(\iota, \mathscr{L}) = \langle \mathtt{C}, \ldots \rangle, \mathscr{L}' \\ \mathit{meth}_{\overline{\psi}}(\mathtt{m}, \mathtt{C}) = \_\ \mathtt{m}(\_\ \overline{\mathtt{x}})\{\mathtt{return}\ \mathtt{e}_0;\}\end{array}}{\_, \mathscr{L}, \overline{\iota}, \iota.\mathtt{m}(\overline{\mathtt{v}}) \longrightarrow \_, \mathscr{L}', \overline{\iota}\iota, \mathtt{return}([^{\overline{\mathtt{x}}}/_{\overline{\mathtt{v}}}, {}^{\mathtt{this}}/_{\iota}]\mathtt{e}_0)}$$

(C-Ret)

$$\_, \mathscr{L}, \overline{\iota}\iota, \mathtt{return}(\mathtt{v}) \longrightarrow \_, \mathscr{L}, \overline{\iota}, \mathtt{v}$$

IFDΔJ: type soundness

Type soundness

Let $\mathtt{L} = (\overline{\psi}, \Phi, \mathtt{DMT}, \Delta, \prec, \mathtt{RG})$ be a well-typed IFDΔJ dynamic SPL. If $\overline{\psi}, \emptyset, \bullet, e$ is the initial state for a valid product $\mathtt{CT}_{\overline{\psi}}$ and $\overline{\psi}, \emptyset, \bullet, e \Longrightarrow^{*} \overline{\psi}', \mathscr{L}', \overline{\iota}, e' \not\longrightarrow$, then e' is either

- `null`, or
- an address ι such that $\mathscr{L}(\iota) = \langle \mathtt{C}, \overline{\mathtt{f}} = \overline{\mathtt{v}} \rangle$ with $\mathtt{C} <:_{\overline{\psi}'} \mathtt{C}_{\mathtt{Main}}$, or
- an expression containing either $\mathtt{null.f}$ or $\mathtt{null.f} = \mathtt{v}$ or $\mathtt{null.m}(\overline{\mathtt{v}})$ for some `f`, `v`, `m`, and $\overline{\mathtt{v}}$.

Some Recent Related Work

- Dynamic classes: runtime updates in distribute OO systems [Johnsen et al., FM 2009]
- Direct semantics for FOP SPL [Apel et al., J. of Automated Software Engineering, 2010]
- Aspect-based dynamic software update extraction [Cech Previtali and Gross, AOSD 2011]
- JavAdaptor [Pukall et al. 2013]

Conclusion

Summary:

- Dynamic Delta-Oriented Programming

Future Work:

- Case studies (application to autonomic systems?)
- A *direct semantics* for Dynamic DOP [Apel et al., J. ASE, 2010]
- Extension to *distribute systems* [Johnsen et al., FM 2009]
- Developing a proof system for Dynamic DOP [Damiani et al., FMSPLE@SPLC 2012]
- A general paradigm for Unanticipated Software Evolution
    - Programming languages where code update is a first class construct