IRIS AperTO

UNIVERSITÀ DEGLI STUDI DI TORINO

AperTO - Archivio Istituzionale Open Access dell'Università di Torino

A core calculus for dynamic delta-oriented programming

This is the author's manuscript

Original Citation:

Availability:

This version is available http://hdl.handle.net/2318/1633408 since 2018-07-25T15:19:51Z

Published version:

DOI:10.1007/s00236-017-0293-6

Terms of use:

Open Access

Anyone can freely access the full text of works made available as "Open Access". Works made available under a Creative Commons license can be used according to the terms and conditions of said license. Use of all other works requires consent of the right holder (author or publisher) if not exempted from copyright protection by the applicable law.

(Article begins on next page)

19 March 2024

IRIS AperTO

UNIVERSITÀ DEGLI STUDI DI TORINO

This is the author's version of the contribution published as:

Damiani, F., Padovani, L., Schaefer, I., Seidl, C. A core calculus for dynamic delta-oriented programming. Acta Informatica (2017).

doi:10.1007/s00236-017-0293-6

The publisher's version is available at:

http://link.springer.com/article/10.1007/s00236-017-0293-6

When citing, please refer to the published version.

The final publication is available at

link.springer.com

Noname manuscript No.
(will be inserted by the editor)

A Core Calculus for Dynamic Delta-Oriented Programming

Ferruccio Damiani · Luca Padovani · Ina Schaefer · Christoph Seidl

the date of receipt and acceptance should be inserted later

Abstract Delta-oriented programming (DOP) is a flexible approach to the implementation of software product lines (SPLs). Delta-oriented SPLs consist of a code base (a set of delta modules encapsulating changes to object-oriented programs) and a product line declaration (providing the connection of the delta modules with the product features). In this paper, we present a core calculus that extends DOP with the capability to switch the implemented product configuration at runtime. A dynamic delta-oriented SPL is a delta-oriented SPL with a dynamic reconfiguration graph that specifies how to switch between different feature configurations. Dynamic DOP supports also (unanticipated) software evolution such that at runtime, the product line declaration, the code base and the dynamic reconfiguration graph can be changed in any (unanticipated) way that preserves the currently running product, which is essential when evolution affects existing features. The type system of our dynamic DOP core calculus ensures that the dynamic reconfigurations lead to type safe products and do not cause runtime type errors.

1 Introduction

A *software product line* (SPL) is a family of software systems with well-defined commonalities and variabilities that are developed by (re)using common artifacts (Pohl et al. [2005]). Many industries have successfully adopted an SPL development approach for building families of related systems with better quality, shorter time-to-market, and lower production costs. Modern software systems tend to be extremely long-lived. Hence, they have to evolve to meet changing user requirements or resource constraints over time. To remain operational over long periods, these systems additionally need to be designed to adapt at runtime

This work has been partially supported by: project HyVar (www.hyvar-project.eu), which has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 644298; by ICT COST Action IC1402 ARVI (www.cost-arvi.eu); by ICT COST Action IC1201 BETTY (http://www.behavioural-types.eu/); and by Ateneo/CSP D16D15000360005 project RunVar.

F. Damiani · L. Padovani
Università di Torino

I. Schaefer · C. Seidl
Technische Universität Braunschweig

due to both reconfiguration and evolution. Conventional (static) SPLs fail to provide mechanisms for addressing these new requirements. Dynamic software product lines (Hallsteinsen et al. [2008], Capilla et al. [2014]) focus on engineering adaptive systems using a dedicated variability model describing all possible configurations a system may adapt to at runtime. *Delta-oriented programming* (DOP) (Schaefer et al. [2010], Bettini et al. [2013b]) is a flexible approach for implementing SPLs that has so far only been used to implement variability which is bound before compile-time. In this paper, we present *dynamic DOP* to realize runtime variability and evolution based on DOP. We provide a formal foundation for dynamic DOP together with a type system ensuring the type safety of dynamic reconfiguration.

A delta-oriented SPL consists of a *code base* comprising a set of delta modules and a *product line declaration* linking delta modules to the product features (Schaefer et al. [2010]). A *delta module* encapsulates modifications to an object-oriented program. A particular product in a delta-oriented SPL is generated by applying the modifications contained in the suitable delta modules to a core program that, without loss of generality, can always be assumed to be empty (Schaefer and Damiani [2010]). A *dynamic* delta-oriented SPL adds to these a *dynamic reconfiguration graph* defining which configurations the system can adapt to at runtime and describing how existing objects need to be reconfigured in case they are instances of classes changed by the reconfiguration. To mitigate the runtime overhead caused by reconfiguration, it would be desirable that existing objects are reconfigured *on demand* only when their fields are accessed or a method is called upon them. Besides reconfiguration at runtime by changing the currently enabled features, dynamic DOP also supports unanticipated evolution by introducing or removing products from the product line and modifying the implementation of existing products. This evolution can be carried out at runtime relying on similar principles as reconfiguration at runtime.

In summary, the contributions of this work are as follows:

- we extend DOP to model dynamic SPLs;
- we define a core calculus that formalizes the operational semantics of dynamic DOP runtime reconfiguration and evolution and that supports lazy object reconfiguration;
- we provide a type system for the dynamic DOP core calculus ensuring that dynamic reconfiguration and evolution leads to type safe products and does not cause runtime type errors.

It is worth observing that the dynamic reconfiguration graph, which is the novel programming construct introduced in this paper, is decoupled from the structure of the code base (i.e., from delta modules). So it could be used in connection with other approaches for implementing SPLs, like, e.g., *Feature-oriented programming* (FOP) (Batory et al. [2004], Kästner et al. [2008])—we refer to Schaefer et al. [2012] for a survey on approaches for implementing SPLs.

The paper is organized as follows. Section 2 introduces dynamic DOP by means of examples. Section 3 recalls the core calculus for DOP called IFΔJ (Bettini et al. [2013b]). Section 4 presents syntax, type system, operational semantics, and type soundness of the core calculus for *dynamic* DOP called IFDΔJ. Section 5 discusses related work. Section 6 concludes by outlining possible directions for future work. The appendix contains the proofs of the main results.

Preliminary versions of the material presented in this paper appeared in (Damiani and Schaefer [2011], Damiani et al. [2012b]). This paper contains new and improved explanations and examples, more details of the formalization, and the proofs of the main results.

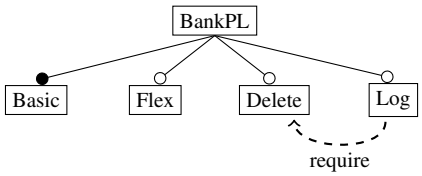


Fig. 1 Feature model for the Bank PL

2 Dynamic Delta-Oriented Programming

To illustrate dynamic delta-oriented programming, we introduce an example of a simple product line of programs for supporting the activities of a bank, which we will call the Bank PL. The example aims at illustrating the main concepts of dynamic delta-oriented programming, rather than at providing a realistic product line case study. We use a *feature model* as variability model for illustrative purposes for our example as presented in Figure 1. A feature model is a compact representation of all permissible configurations of an SPL that describes configurable functionality on a conceptual level in terms of *features* that can be selected or deselected: an optional feature (hollow circle) may be selected or deselected and a *mandatory* feature (filled circle) has to be selected. The root feature of a feature model is implicitly regarded as being mandatory. Furthermore, if a child feature is selected, then its parent feature has to be selected as well. In addition, cross-tree constraints (dashed arrows) may be used to specify the dependency of one on another. A valid *product* of a feature model obeys all configuration rules imposed by the feature model and its cross-tree constraints. Both our example SPL and the IFDΔJ core calculus presented in Sections 3 and 4 use a JAVA-like syntax. In order to improve readability, in the example we use a richer syntax, including `void`, the primitive types `int` and `boolean`, arrays, the shortcut syntax for operations on strings, conditional and loop statements, and the sequential composition. Encoding in IFDΔJ syntax is straightforward—see Bettini et al. [2013b] for examples. The products in the Bank PL represent the actual configurations a concrete bank can decide to operate in. These configurations are described as the set of selected features Basic, Flex, Delete and Log. The feature Basic is mandatory and comprises the fundamental functionality of a bank: the ability to execute commands (by means of a class `Controller`) like creating a new account (by means of a class `AccountScanner`), retrieving an account, updating the balance of an account and deducting the transaction fee. The feature Flex is optional and provides the functionality for managing an arbitrary number of accounts. The feature Delete is optional and provides the functionality for deleting an account. The feature Log is also optional and ensures that all the operations performed on an account are logged. In addition, the feature Log requires the presence of the feature Delete.

A dynamic delta-oriented product line consists of a *code base*, a *product line declaration* and a *dynamic reconfiguration graph*, which we describe in the rest of this section.

2.1 Product-Line Code Base

When using DOP for SPL development, a product line code base consists of a set of *delta modules*, which are containers for a sequence of modifications to an object-oriented program. The modifications may add, remove or modify classes. Modifying a class means to change its super class, to add or to remove fields or methods or to modify methods.

The modification of a method can either replace the method body by another implementation, or wrap the existing method using the `original` construct (similar to the `Super()` call in AHEAD (Batory et al. [2004])). The `original` construct expresses a call to the method with the same name before the modifications and is bound at the time the product is generated. Before or after the `original` construct, other statements can be introduced to wrap the existing method implementation. In addition to proactive product line development (building a product line entirely anew), DOP also supports extractive product line development (Krueger [2002]), starting from an existing legacy product (Schaefer and Damiani [2010]).

Listing 1 contains the code base for the Bank PL. The delta module `DBasic` introduces, by modifying the empty program, the code of (what we assume to be) an existing legacy product, realizing the feature Basic. The feature Basic is implemented by the classes `Account`, `Bank`, `AccountScanner`, `Controller` and `Main`.

- The class `Account`, which represents a bank account, contains a `balance` field and an `owner` field, an `update` method for manipulating the balance, and a `toString` method to produce a textual representation of the data of an account.
- The class `Bank`, which represents the bank, contains the field `accountAt` for storing the accounts, the field `next` for storing the identifier of the next account to be created (the identifier of each account is its position in the array `accountAt`), the field `fee` for storing the fee to be charged on an account for each update operation, a method `init` for initializing a `Bank` object after creation, a method `isValid` for checking whether a given account number is valid (i.e., associated with an existing account), a method `addAccount` for adding a new account (when the `accountAt` array is full, the new account is not added and the value `-1` is returned), a method `retrieveAccount` to retrieve an account from its identifier (when there is no account with such an identifier, the value `null` is returned), and a method `update` for manipulating the balance of an account given its identifier (when there is no account with such an identifier, the value `false` is returned).
- The class `AccountScanner`, whose implementation details are omitted, defines a method `nextAccount`, which parses the data of an account from an input stream.
- The accounts of the bank are manipulated through the class `Controller`, which provides a method `execute` for executing the commands, represented by the strings `"add"` (for adding a new account), `"retrieve"` (for printing the details of an existing account), and `"update"` (for updating the balance of an account).
- The class `Main` provides a method `main`, which starts the application by creating a scanner, an account scanner, a bank and a controller for the management of the bank accounts, after which it loops forever by reading commands from the input stream and passing them to the controller.

The delta module `DFlex` implements the feature Flex by modifying the class `Bank`. It modifies the method `init` so that, when a new `Bank` object is initialized, the field `fee` is set to 3 (a “flexible bank” requires an increased transaction fee in order to cover the costs brought by the flexibility). It further modifies the method `addAccount` so that, when the array `accountAt` is full, its size is doubled.

The delta module `DDelete` implements the feature Delete. It modifies the class `Bank` by adding the method `deleteAccount` (for deleting an account) and modifying the method `isValid` (for ensuring that deleted accounts are not valid). It also modifies the `Controller` class so that the command `delete` is accepted.

```
delta DBasic {
  adds class Account {
    int balance; String owner;
    void update(int amount) { balance = balance + amount; }
    String toString() { return owner + ":" + balance; }
  }
  adds class Bank {
    Account[] accountAt; int next; int fee;
    void init(int n) { accountAt = new Account[n]; fee = 2; }
    boolean isValid(int id) { return ((0 <= id) && (id <= next−1)); }
    int addAccount(Account a) { if (next == accountAt.length) { return −1; } accountAt[next] = a; return next++; }
    Account retrieveAccount(int id) { if (!(isValid(id))) { return null; } return accountAt[id]; }
    boolean update(int id, int x) { if (!(isValid(id))) { return false; } accountAt[id].update(x−fee); return true; }
  }
  adds class AccountScanner ... // parses data of Account type
  adds class Controller {
    Scanner s; AccountScanner as; Bank b;
    void init(Scanner scanner, AccountScanner accounScanner, Bank bank) { s=scanner; as=accountScanner; b=bank; }
    void execute(String command) {
      if (command.equals("add")) { Account a = as.nextAccount(); System.out.println(b.addAccount(a)); return; }
      if (command.equals("retrieve")) { int id = s.nextInt(); System.out.println(b.retrieveAccount(id)); return; }
      if (command.equals("update")) { int id = s.nextInt(); int amount = s.nextInt(); b.update(id,amount); return; } }
  }
  adds class Main {
    void main() {
      Scanner scanner = new Scanner(System.in);
      AccountScanner accountScanner = new AccountScanner(System.in);
      Bank bank = new Bank().init(100);
      Controller controller = new Controller().init(scanner,accountScanner,bank);
      while (true) { String command = scanner.next(); controller.execute(command); } }
  }
}

delta DFlex {
  modifies class Bank {
    modifies void init(int n) { accountAt = new Account[n]; fee = 3; }
    modifies int addAccount(Account a) {
      if (next == accountAt.length) { Account[] newAccountAt = new Account[next∗2];
                                      for (int i= 0; i < next; i++) { newAccountAt[i] = accountAt[i]; } }
      accountAt = newAccountAt; return original(a); }
  }
}

delta DDelete {
  modifies class Bank {
    modifies boolean isValid(int id) { return (original(id) && (accountAt[id]!=null)); }
    adds boolean deleteAccount(int id) { if (isValid(id)) { accountAt[id] = null; return true; } else return false; }
  }
  modifies class Controller {
    modifies void execute(String command) {
      original(command);
      if (command.equals("delete")) { int id = s.nextInt(); System.out.println(a.delete(id)); return; } }
  }
}

delta DLog {
  modifies class Account {
    adds String log;
    modifies void update(int x) { log = log + "[update:" + x + "]"; original(x); }
    adds String getLog() { return log; }
  }
  modifies class AccountScanner ... // to initialize the field log
  modifies class Controller {
    modifies void execute(String command) {
      original(command);
      if (command.equals("getLog")) { int id = s.nextInt(); Account a = b.retrieveAccount(id); if (a == null) {
                                      System.out.println(null); } else { System.out.println(a.getLog()); } return; } }
  }
}
```

Listing 1: Code base of the Bank PL

```
features Basic, Flex, Delete, Log
configurations Basic & (Log -> Delete)
deltas
  { DBasic }
  { DFlex when Flex, DDelete when Delete }
  { DLog when Log }
```

Listing 2: Declaration of the Bank PL

The delta module `DLog` implements the feature `Log`. It modifies the class `Account` by introducing the field `log` (for recording the operations executed on the account), modifying the `update` method accordingly, and adding the method `getLog` (for returning the log). It also modifies the `AccountScanner` class to ensure that the field `log` is properly initialized, and further modifies the `Controller` class so that the command `getLog` is accepted.

2.2 Product-Line Declaration

The delta modules of a product line code base describe which modifications to perform on an object-oriented program. To specify a full SPL, it is further necessary to define which features exist, which configurations of features are considered valid and which delta modules are associated with which features. We allow specifying this information in a *product-line declaration*. Listing 2 shows the product line declaration for the Bank PL.

In the example, the essential elements of a product-line declaration are demonstrated: Line 1 defines the principally available features of the SPL. Line 2 represents all valid configurations in terms of a propositional formula over features[1]. Lines 3–6 associate each delta module with an *activation condition* in a `when` clause to specify that the delta module is only applied if the specified features are part of the configuration.

Typically, more than one delta module must be used for the generation of a product. However, the order in which the delta modules are applied may not be chosen arbitrarily if multiple delta modules modify overlapping parts of the object-oriented program. To accommodate for that case, we allow the specification of *application orders* on delta modules to state that a certain group of delta modules may only be applied after another group of delta modules. Groups of delta modules are defined by a list of delta modules enclosed by { .. } as presented in Listing 2, e.g., in Line 5. Within each group, delta modules may be applied in an arbitrary order.

To obtain a product for a particular configuration, first, those delta modules are collected that have a valid activation condition according to the selected features. After that, a sequence for the delta modules is established according to the application order before the modifications specified in the delta modules are applied incrementally according to the established order. The first delta module is applied to the empty product. The modifications of a delta module are applicable to a (possibly empty) product if each class to be removed or modified exists and, for every modified class, if each method or field to be removed exists, if each method to be modified exists and has the same signature as the modified method, and if each class, method or field to be added does not exist. During the generation of a product, every delta module must be applicable. Otherwise, the generation of the product fails. In particular, if applied to the empty product, the first delta module can only contain additions.

[1] It is generally not possible to enumerate all possible configurations due the sheer number in any non-trivial SPL. However, there are other ways of representing valid configurations of an SPL, e.g., see (Batory [2005]) for other representations.

Listing 3 presents the product generated when all the features (Basic, Flex, Delete and Log) are selected. Note that a method-modify operation that uses the `original` construct adds a new method with a fresh name that is used (instead of `original`) in the body of the modified method in the generated product. The name of the new method is denoted by $\texttt{m\$}\delta$, where `m` is the name of the modified method and δ is the name of the delta module that contains the method-modify operation (cf. methods `update$Dlog`, `isValid$Delete`, `addAccount$DFlex`, `execute$DLog` and `execute$DDelete` in Listing 3).

2.3 Product Line Dynamic Reconfiguration Graph

Reconfiguration alters the currently active configuration by enabling or disabling certain features to realize different functionality. A reconfiguration may be prompted manually or by any other external event. The change of feature in the active configuration also affects the code realizing that configuration: When reconfiguration is performed at compile time, the respective source code has to be re-assembled and deployed. However, when reconfiguration is performed during runtime, not only the code has to be re-assembled but also the objects in the heap must be updated accordingly if their respective classes are affected by the reconfiguration. A class `C` is *affected* by a reconfiguration if it is *recoded* or *reallocated*:

- A class `C` is *recoded* if its *code* (or that of one of its superclasses) is changed. This means that fields and methods may be added, removed, or modified.
- A class `C` is *reallocated* if its *object instances* are changed. Note that, if the reconfiguration changes the value of some fields of `C`, then `C` is reallocated even if it is not recoded.

The notion of affected class (formalized in Section 4.1) will be used to avoid inconsistent behavior: switching to the new feature configuration is enabled only if all methods that are on the call stack have receivers whose classes are unaffected by the reconfiguration. Such restriction applies also to those evolutions that change the implementation of individual features. Note that the classes that are recoded and/or reallocated by a reconfiguration or evolution can be determined statically.

The dynamic reconfiguration graph is a directed graph whose nodes are (a subset of) the configurations and each edge is labeled by a set of *object reconfiguration clauses*. When two feature configurations $\overline{\varphi}$ and $\overline{\psi}$ are *adjacent* in the dynamic reconfiguration graph, then it is possible to change the feature configuration of the currently running product from $\overline{\varphi}$ to $\overline{\psi}$. The object reconfiguration clause that labels the edge from $\overline{\varphi}$ to $\overline{\psi}$ specifies how to reconfigure the objects on the heap that are instances of the affected classes. An object reconfiguration clause `OR` has the following syntax

$$\texttt{C} \rightarrow \texttt{C}'\{\texttt{pre: } \overline{\texttt{A y} = \ldots}\texttt{; post: } \overline{\texttt{B z} = \ldots}\texttt{; } \overline{\texttt{this.f} = \texttt{z}'}\texttt{;}\}$$

which specifies the new class $\texttt{C}'$ of an object of class `C` and the rearrangements of its fields in the new configuration (each class has an implicit default constructor that is used for creating the new object). The pre-reconfiguration assignments $\overline{\texttt{A y} = \ldots}$, where `A` denotes a type and `y` denotes a local variable, are used to retrieve (from the heap before the reconfiguration) values that are necessary for the object reconfiguration. The post-reconfiguration assignments $\overline{\texttt{B z} = \ldots}$, where `B` denotes a type and `z` denotes a local variable, take care of the proper migration from the old to new feature configuration of classes. Namely, either the content of a variable $\texttt{y}' \in \overline{\texttt{y}}$ or the address of a newly created object is assigned to each of the variables $\overline{\texttt{z}}$, whose types refer to the new configuration. Unmodified fields (i.e., fields that occur both

```
class Account {
  int balance;
  String owner;
  String log;
  String toString() { return owner + ":" + balance; }
  void update(int x) { log = log + "[update:" + x + "]"; update$DLog(x); }
  void update$Dlog(int amount) { this.balance = this.balance + amount; }
  String getLog() { return log; }
}

class Bank {
  Account[] accountAt;
  int next;
  int fee;
  void init(int n) { accountAt = new Account[n]; fee = 3; }
  boolean isValid(int id) { return (isValid$Delete(id) && (accountAt[id]!=null)); }
  boolean isValid$Delete(int id) { return ((0 <= id) && (id <= next−1)); }
  int addAccount(Account a) {
    if (next == accountAt.length) { Account[] newAccountAt = new Account[next∗2];
                    for (int i= 0; i < next; i++) { newAccountAt[i] = accountAt[i]; } }
    accountAt = newAccountAt;
    return addAccount$DFlex(a); }
  int addAccount$DFlex(Account a) { if (next == accountAt.length) { return −1; }  accountAt[next] = a;
    return next++; }
  Account retrieveAccount(int id) { if (!(isValid(id))) { return null; } return accountAt[id]; }
  boolean update(int id, int x) { if (!(isValid(id))) { return false; } accountAt[id].update(x−fee); return true; }
  boolean deleteAccount(int id) { if (isValid(id)) { accountAt[id] = null; return true; } else return false; }
}

class AccountScanner ... // parses data of Account type

class Controller {
  Scanner s;
  AccountScanner as;
  Bank b;
  void init(Scanner scanner, AccountScanner accounScanner, Bank bank) { s=scanner; this as=accountScanner;
    b=bank; }
  void execute(String command) {
    execute$DLog(command);
    if (command.equals("getLog")) { int id = s.nextInt();
                                    Account a = b.retrieveAccount(id);
                                    if (a == null) { System.out.println(null); }
                                      else { System.out.println(a.getLog()); }
                                    return; }
  }
  modifies void execute$DLog(String command) {
    execute$DDelete(command);
    if (command.equals("delete")) { int id = s.nextInt(); System.out.println(a.delete(id)); return; }
  }
  void execute$DDelete(String command) {
    if (command.equals("add")) { Account a = as.nextAccount(); System.out.println(b.addAccount(a)); return; }
    if (command.equals("retrieve")) { int id = s.nextInt(); System.out.println(b.retrieveAccount(id)); return; }
    if (command.equals("update")) { int id = s.nextInt(); int amount = s.nextInt(); b.update(id,amount); return; }
  }
}

class Main {
  void main() {
    Scanner scanner = new Scanner(System.in);
    AccountScanner accountScanner = new AccountScanner(System.in);
    Bank bank = new Bank().init(100);
    Controller controller = new Controller().init(scanner,accountScanner,bank);
    while (true) { String command = scanner.next(); controller.execute(command); }
  }
}
```

Listing 3: Product generated when the features Basic, Flex, Delete and Log are selected

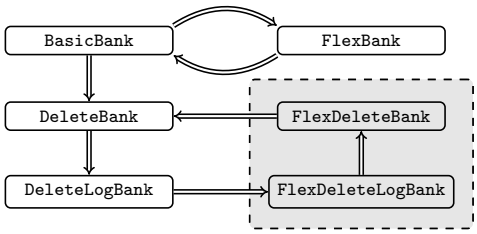


Fig. 2 Dynamic reconfiguration graph of the Bank PL (abstract graphical representation)—the gray box highlights the part of the graph that is the same as in Fig. 4

in C and C′ with the same name and type) are copied by default and the remaining fields are initialized to the default values associated to their type (as in JAVA). Then, the assignments $\overline{\texttt{this.f} = \texttt{z}'}$ allow the programmer to update the value of the (unmodified and new) fields of the reconfigured object of class C′ by using variables $\texttt{z}' \in \overline{\texttt{z}}$. Note that the fields of C′ are not visible in the pre-reconfiguration assignments and the fields of C are not visible in the post-reconfiguration assignments. This makes it possible to handle reconfigurations where C and C′ contain two fields with the same name and different types.

Empty `pre:` and `post:` clauses are omitted. If the classes C and C′ are equal, only the name of the affected class C is written, instead of $\texttt{C} \to \texttt{C}$. Moreover, if C is (recoded and) not reallocated, then the entire body $\{...\}$ of the reconfiguration clause is omitted—note that specifying the *empty body* has a different meaning from specifying *no body* (for instance, an empty body must be specified when an object is reconfigured by dropping some of its fields and preserving the value of the fields that are not dropped).

Each edge of the dynamic reconfiguration graph is labeled by an object reconfiguration clause `OR`. The operational semantics (Section 4.3) uses the clause to check that the reconfiguration associated to a given edge can be safely performed in a given state of the computation. To prevent reconfiguration of an object while a method is currently executed on it, an object reconfiguration clause must be present for each affected class.

Listing 4 describes the dynamic reconfiguration graph for the Bank PL (the product line declaration is in Listing 2). The first part of the description, beginning with the keyword **nodes**, declares the nodes of the graph by associating each name of a node with a distinct configuration, e.g., `FlexDeleteBank` with the configuration {Basic, Flex, Delete}.

The second part of the description, beginning with the keyword **edges**, declares the edges of the graph, which specify the possible runtime reconfigurations, e.g., between `BasicBank` and `FlexBank` (and vice versa). Figure 2 depicts (an abstract graphical representation of) the dynamic reconfiguration graph described in Listing 4.

- The edge `BasicBank=>FlexBank` affects the class `Bank` for which it contains an object reconfiguration clause. The class `Bank` is both recoded (because the method `addAccount` is modified) and reallocated (because the value of the `fee` field is changed). The reallocation is needed because a "flexible bank" requires an increased transaction fee in order to cover the costs brought by the flexibility. This is achieved by inserting suitable operations in the body of the object reconfiguration clause.
- The edge `BasicBank=>DeleteBank` affects the classes `Bank` and `Controller`. Therefore, it contains an object reconfiguration clause for each of them. Both the classes `Bank` and `Controller` are recoded and not reallocated. Hence, the object reconfigura-

```
nodes
  BasicBank = Basic;
  FlexBank = Basic, Flex;
  DeleteBank = Basic, Delete;
  DeleteLogBank = Basic, Delete, Log;
  FlexDeleteBank = Basic, Flex, Delete;
  FlexDeleteLogBank = Basic, Flex, Delete, Log;
edges
  BasicBank => FlexBank {
    Bank { pre:  int tmpFee = this.fee;
           post:  this.fee = tmpFee + 1; }
  }
  FlexBank => BasicBank {
    Bank { pre: int  tmpFee = this.fee;
           post:  this.fee = tmpFee − 1; }
  }
  BasicBank => DeleteBank {
    Bank,
    Controller
  }
  DeleteBank => DeleteLogBank {
    Account { post:  this.log = ""; },
    Controller,
    AccountScanner
  }
  DeleteLogBank => FlexDeleteLogBank {
    Bank { pre:  int tmpFee = this.fee;
           post:  this.fee = tmpFee + 1; }
  }
  FlexDeleteLogBank => FlexDeleteBank {
    Account { },
    Controller,
    AccountScanner
  }
  FlexDeleteBank => DeleteBank {
    Bank { pre:  int tmpFee = this.fee;
           post:  this.fee = tmpFee − 1; }
  }
```

Listing 4: Dynamic reconfiguration graph of the Bank PL

tion clauses have no body. Classes `Account`, `AccountScanner` and `Main` are unaffected. Hence, there are no object reconfiguration clauses for them.

- The edge `DeleteBank=>DeleteLogBank` affects the classes `Account`, `AccountScanner` and `Controller` and contains object reconfiguration clauses for these classes. Class `Account` is both recoded and reallocated. Its object reconfiguration clause specifies that the new field `log` is initialized to the empty string (the other fields, which are not changed by the reconfiguration, are implicitly copied). Since the classes `AccountScanner` and `Controller` are recoded (and not reallocated) their object reconfiguration clauses have no body. Class `Main` is unaffected, so there is no object reconfiguration clause for it.
- The edge `FlexDeleteLogBank=>FlexDeleteBank` recodes and reallocates the `Account` class, it recodes (but does not reallocate) classes `AccountScanner` and `Controller`, and it does not affect classes `Bank` and `Main`. Note that the object reconfiguration clause for the class `Account` has an empty body so that the `log` field is dropped and the other fields are preserved.
- The remaining transitions are similar in nature to the ones described above.

A reconfiguration $\overline{\varphi}$=>$\overline{\psi}$ is *enabled* when there is no running method invoked on an instance of a class affected by the reconfiguration. As already pointed out at beginning of Section 2.3, the switch to a new feature configuration is enabled only if all methods that are

on the call stack have receivers whose classes are unaffected by the reconfiguration. This condition is sufficient to enforce type soundness.

For instance, when the running product is in any configuration $\overline{\varphi}$ of the Bank PL (cf. Listing 4) and the first statement in the body of method `main` of class `Main` is being executed or the statement `String command = scanner.next();` in the body of the **while** loop of the same method is being executed (cf. Listing 1), all the reconfigurations to any node adjacent to $\overline{\varphi}$ in the dynamic reconfiguration graph (cf. Figure 2) are enabled. Instead, when the method `execute` of class `Controller` is the last invoked and currently executing method, only the reconfigurations `BasicBank=>FlexBank`, `FlexBank=>BasicBank` and `DeleteLogBank=>FlexDeleteLogBank` are enabled.

The core calculus for dynamic DOP (Section 4) models a lazy update of the heap so that each object is reconfigured only if and when the running product accesses it.[2] To model lazy heap update, a queue of pending reconfiguration operations is maintained while the heap is partitioned in regions. If there are n pending reconfiguration operations, then there are $n+1$ regions (numbered form 0 to n). The 0-th region contains the objects that have not yet been reclassified by any of the pending operations, the 1-st region contains objects that have passed the first pending operation and so on. The n-th region contains the objects that have passed all pending operations. When the 0-th region becomes empty, it is destroyed and the first pending operation is removed from the queue. Whenever the running product requires accessing an object that is not in the n-th region, all the pending reconfigurations on the object are performed and the object is moved in the n-th region (which may in turn imply the reconfiguration of other objects).

2.4 Evolving Dynamic Delta-Oriented SPLs

Reconfiguring an SPL means selecting a new valid configuration of features and activating it. However, over the course of time, new or changed requirements on the SPL may cause the SPL itself to change, e.g., by adding new features and delta modules, by deleting old features and delta modules, or by making existing features optional or mandatory. While this distinction of reconfiguration and evolution is essential at a conceptual level (Seidl et al. [2014]), at the implementation level of source code, changes associated with both reconfiguration and evolution manifest as modifications of the source code, in our case during runtime. As a consequence, the product line declaration, the dynamic reconfiguration graph, or the code base may have to be replaced. At runtime, these artifacts can safely be replaced if the following two conditions hold:

1. *Currently running product is preserved*: For the current configuration, the new code base and the new product line declaration describe the same product (obtained by applying the same delta modules in the same order).
2. *Pending reconfigurations are preserved*: In the new reconfiguration graph, the edges associated with the pending reconfigurations/evolutions are unchanged and all the products reached by these edges are preserved by the new code base and the new product line declaration. Otherwise, application of the pending reconfigurations/evolutions would have to be performed before the update, which might take indefinite amounts of time if a method remains on the call stack.

In the rest of this section, we illustrate three different evolution examples for the Bank PL.

[2] The formalization performs the least amount of necessary updates. An implementation may choose to follow a more eager strategy.

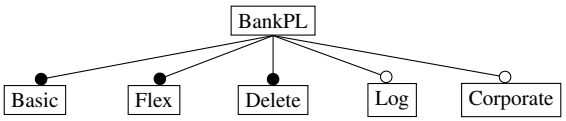


Fig. 3 Feature model for the evolved Bank PL

Example 1 Assume that the Bank PL has to evolve to meet changed or entirely new requirements: The features Flex and Delete are made mandatory, which consequentially discards the products `BasicBank`, `FlexBank`, `DeleteBank`, and `DeleteLogBank`). A new optional feature Corporate is introduced for opening a new kind of account with reduced fees, reserved to the employees of corporations that signed an agreement with the bank, which consequentially creates the new products `FlexDeleteCorporateBank` and `FlexDeleteLogCorporateBank`. Adding the feature Corporate is challenging as the Bank PL was designed without foreseeing the possibility of having accounts of different kinds with different transaction fees. In particular, the deduction of transaction fees which are the same for all the accounts is built into the class `Bank`, which is introduced by the delta module `DBasic` for the feature Basic (cf. Figure 1). The feature model of the evolved Bank PL is shown in Figure 3.

To evolve the implementation of the Bank PL, the following changes have to be made:

- Modify the code base by adding the delta modules `DCorporate` and `DCorporateLog` as illustrated in Listing 5. `DCorporate` moves the logic for deducting fees from the class `Bank` to the class `Account`, introduces the class `CorporateAccount` for the new kind of account and modifies the class `AccountScanner` to parse corporate accounts. `DCorporateLog` modifies the class `Account` to ensure that fees are logged.
- Modify the product-line declaration so that discarded products may no longer be created and all new products are supported. The resulting declaration is shown in Listing 6.
- Modify the dynamic reconfiguration graph by: *(i)* removing the nodes for the dropped products together with their incident edges; and *(ii)* adding the nodes for the new products and the new edges. The resulting graph is illustrated in Listing 7 and its abstract graphical representation is presented in Figure 4. Note that, in the course of evolution, the edge `FlexDeleteLogBank=>FlexDeleteBank` remains unchanged (cf. Listing 4 and Figure 2), while the other three edges are new.
  - The edge `FlexDeleteBank=>FlexDeleteCorporateBank` specifies that the class `Bank` is both recoded and reallocated. Classes `Account` and `AccountScanner` are recoded (but not reallocated).
  - The edge `FlexDeleteCorporateBank=>FlexDeleteLogCorporateBank` is largely similar to the edge `DeleteBank=>DeleteLogBank` in Listing 4, the only difference is that it also contains the object reconfiguration clause for the class `CorporateAccount`, which is both recoded and reallocated.
  - The edge `FlexDeleteLogCorporateBank=>FlexDeleteLogBank` contains an example of an object reconfiguration clause where objects are reconfigured to be instances of a class with a different name. In particular, all objects of the class `CorporateAccount` are reconfigured to new objects belonging to the class `Account` due to the fact that the class `CorporateAccount` is dropped by the reconfiguration. Furthermore, the change of account type is logged. The other object reconfiguration clauses specify that the classes `Account` and `AccountScanner` are recoded (but not reallocated).

When the running program conforms to one of the two configurations `FlexDeleteBank` or `FlexDeleteLogBank`, both the product line declarations in Listing 2 and Listing 6 de-

```
delta DCorporate {
  modifies class Account {
    adds void chargeFee() { balance = balance − getFee(); }
    adds int getFee() { return 3; }
  }
  adds class CorporateAccount extends Account {
    String corporation;
    modifies toString() { return corporation + "," + original(); }
    int getFee() { return 1; }
  }
  modifies class Bank{
    removes int fee;
    modifies void init(int n) { accountAt = new Account[n]; }
    modifies boolean update(int id, int x)
      { if (!(isValid(id))) { return false; }  accountAt[id].update(x);  accountAt[id].chargeFee();  return true; }
  }
  modifies class AccountScanner ... // to parse corporate accounts
}

delta DCorporateLog {
  modifies class Account {
    modifies void chargeFee() { original();  log = log + "[fee:" + getFee() + "]"; }
  }
}
```

Listing 5: Delta modules for the evolved Bank PL

```
features Basic, Flex, Delete, Log, Corporate
configurations Basic & Delete & Flex
deltas
  { DBasic }
  { DFlex,
    DDelete }
  { DLog when Log }
  { DCorporate when Corporate }
  { DCorporateLog when Corporate & Log }
```

Listing 6: Declaration of the evolved Bank PL

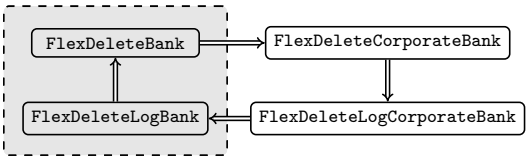


Fig. 4 Dynamic reconfiguration graph of the evolved Bank PL (abstract graphical representation)—the gray box highlights the part of the graph that is the same as in Fig. 2

scribe the same product by applying the same delta modules in the same order. Therefore, when either there are no pending reconfigurations or the only pending reconfiguration is `FlexDeleteLogCorporateBank=>FlexDeleteLogBank`[3] (which is preserved), the product line declaration, code base, and dynamic reconfiguration graph of the Bank PL can safely evolve as described above.

[3] This implies that the running program conforms to configuration `FlexDeleteLogBank`.

```
nodes
  FlexDeleteBank = Basic, Flex, Delete;
  FlexDeleteLogBank = Basic, Flex, Delete, Log;
  FlexDeleteCorporateBank = Basic, Flex, Delete, Corporate;
  FlexDeleteLogCorporareBank = Basic, Flex, Delete, Corporate, Log;
edges
  FlexDeleteBank => FlexDeleteLogBank {
    Account { post: this.log = ""; },
    Controller,
    AccountScanner
  }
  FlexDeleteLogBank => FlexDeleteBank {
    Account { },
    Controller,
    AccountScanner
  }
  FlexDeleteBank => FlexDeleteCorporateBank {
    Account,
    Bank { },
    AccountScanner
  }
  FlexDeleteCorporateBank => FlexDeleteLogCorporateBank {
    Account { post: this.log = ""; },
    CorporateAccount { post: this.log = ""; },
    Controller,
    AccountScanner
  }
  FlexDeleteLogCorporateBank => FlexDeleteLogBank {
    Account,
    CorporateAccount -> Account { pre: String tmpLog = this.log;
                                  post: this.log = tmpLog + "[Till now it was a Corporate Account]"; },
    Bank { post: this.fee = 3; }
    AccountScanner
  }
```

Listing 7: Dynamic reconfiguration graph of the evolved Bank PL

In the previous example, the Bank PL evolved by adding features and delta modules. The following example sketches an evolution that is performed by dropping features and delta modules.

Example 2 Assume that the evolved Bank PL illustrated in Example 1 has to evolve further by "disabling" the feature `Log` (i.e., by "disabling" any reconfiguration involving the products `FlexDeleteLogBank` and `FlexDeleteLogCorporateBank`) and by introducing a new edge that specifies the direct reconfiguration from the product `FlexDeleteCorporateBank` to the product `FlexDeleteBank`.

Whenever the running program conforms to one of the configurations `FlexDeleteBank` or `FlexDeleteCorporateBank` and either there are no pending reconfigurations or the only pending reconfiguration is `FlexDeleteBank=>FlexDeleteCorporateBank`[4] (which is preserved), this evolution can be performed safely by modifying the dynamic reconfiguration graph of the evolved Bank PL: The edge `FlexDeleteBank=>FlexDeleteCorporateBank` is preserved and an edge `FlexDeleteCorporateBank=>FlexDeleteBank` is added as illustrated in Listing 8.

Moreover, although it is not necessary to achieve the desired behavior (cf. the explanation at the beginning of the example), the following changes may be performed:

– The delta modules `DLog` (in Listing 1) and `DCorporateLog` (in Listing 5) can be dropped from the code base.

[4] This implies that the running program conforms to configuration `FlexDeleteCorporateBank`.

```
FlexDeleteCorporateBank => FlexDeleteBank {
  Account,
  CorporateAccount -> Account { },
  Bank { post: this.fee = 3; }
  AccountScanner
}
```

Listing 8: Additional reconfiguration edge `FlexDeleteCorporateBank=>FlexDeleteBank` introduced by evolution

```
features Basic, Flex, Delete, Corporate
configurations Basic & Delete & Flex
deltas
  { DBasic }
  { DFlex,
    DDelete }
  { DCorporate when Corporate }
```

Listing 9: Declaration of the further evolved Bank PL

- The product line declaration can be changed as in Listing 9.
- The nodes `FlexDeleteLogBank` and `FlexDeleteLogCorporateBank` can be dropped from the dynamic reconfiguration graph.

The last example of evolution replaces a delta module by two new delta modules that modularize and improve the changes described by the original delta module.

Example 3 Assume that the Bank PL has to evolve further to support the runtime reconfiguration from configuration `FlexDeleteLogBank` to configuration `FlexBank`. Adding this runtime reconfiguration raises a subtle problem due to two reasons:

- When the running product is in a configuration that includes the feature Delete, it is possible to delete accounts. This, in turn, sets elements of the array `accountAt` with index i to `null` such that $0 \leq i < \texttt{next}$.
- All the configurations that do not include the feature Delete assume that the class `Bank` satisfies the invariant $(0 \leq i < \texttt{next}) \Rightarrow (\texttt{accountAt}[i] \neq \texttt{null})$. Due to this reason, they implement a variant of the method `isValid` that does not explicitly check for deleted (and therefore invalid) account numbers as it lacks the condition `accountAt[i]!=null`.

Therefore, a runtime reconfiguration from a configuration including the feature Delete to a configuration that does not include the feature `Delete` may lead to an inconsistent state. However, this problem can be remedied by using a two-step evolution process.

In the first evolution step, which can be performed when the running product is in configuration `FlexDeleteLogBank` and there are no pending reconfigurations, the Bank PL is evolved into an intermediate product line by the following operations:

- Introduce the new delta modules `DCheck` and `DDelete1` (shown in Listing 11), which modularize and improved the change described by the delta module `DDelete`—namely the modification of the method `isValid` is isolated into the delta module `DCheck` and improved in order to ensure that the check `accountAt[i]!=null` is performed;
- Modify the product line declaration of Listing 2 to create the one shown in Listing 12:
  - Introduce the new feature `Delete1`, which will replace the feature Delete in the final product line.

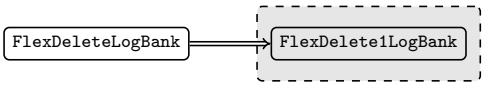


Fig. 5 Dynamic reconfiguration graph of the intermediate Bank PL (abstract graphical representation)—the gray box highlights the part of the graph that is the same as in Fig. 6

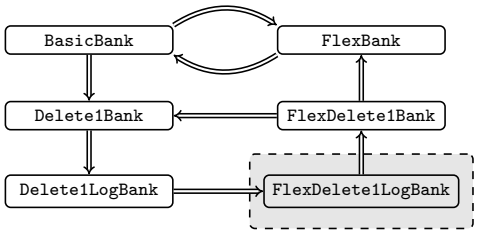


Fig. 6 Dynamic reconfiguration graph of the final Bank PL (abstract graphical representation)—the gray box highlights the part of the graph that is the same as in Fig. 5

```
FlexDeleteLogBank => FlexDelete1LogBank {
  Bank { }
}
```

Listing 10: Additional reconfiguration edge `FlexDeleteLogBank=>FlexDelete1LogBank` introduced by evolution

  - Replace each product that does not include the feature Delete with a variant of the product that implements a variant of the method `isValid` that performs the check `accountAt[i]!=null` by utilizing the delta module `DCheck`.
  - Replace each product that includes the feature Delete by an identical product that has the feature Delete1 as substitute and is implemented by utilizing the delta modules `DCheck` and `DDelete1` as substitute for the delta module `DDelete`.
  - Reintroduce the product for the feature configuration {Basic,Flex,Delete,Log}.
- Modify the dynamic reconfiguration graph of Listing 4 and Figure 2 to create the one presented in Figure 5: Keep the node `FlexDeleteLogBank`. Drop all other nodes and all edges. Introduce the node `FlexDelete1LogBank` for the configuration {Basic, Flex, Delete1, Log} and the edge `FlexDeleteLogBank=>FlexDelete1LogBank` as illustrated in Listing 10.

Once the first evolution step has been performed, the running product is in configuration `FlexDeleteLogBank`. Therefore, when the `main` method of class `Main` is the only method in the call stack, the reconfiguration `FlexDeleteLogBank => FlexDelete1LogBank` is enabled an can be performed.

In the second evolution step, which can be performed when the running product is in configuration `FlexDelete1LogBank`, the intermediate Bank PL is evolved into the final Bank PL by the following operations:

- Drop the delta module `DDelete`.
- Modify the product line declaration produced by the first evolution step to result in the declaration presented in Listing 14 by the following two operations:
  - Drop the feature Delete.

```
delta DCheck {
  modifies class Bank {
    modifies boolean isValid(int id) { return (original(id) && (accountAt[id]!=null)); }
  }
}

delta DDelete1 {
  modifies class Bank {
    adds boolean deleteAccount(int id) {  if (isValid(id)) { accountAt[id] = null; return true; } else return false;  }
  }
  modifies class Controller {
    modifies void execute(String command) {
      original(command);
      if (command.equals("delete")) { int id = s.nextInt();  System.out.println(a.delete(id));  return; } }
  }
}
```

Listing 11: Delta modules `DCheck` and `Delete1`

```
features Basic, Flex, Delete, Log, Corporate, Delete1
configurations ((Basic & (Log −> Delete1)) & !Delete)
               | ((Basic & Flex &  Delete & Log & !Corporate) & !Delete1)
deltas
  { DBasic }
  { DCheck when !Delete}
  { DFlex when Flex,
    DDelete when Delete,
    DDelete1 when Delete1 }
  { DLog when Log }
```

Listing 12: Declaration of the intermediate Bank PL

```
FlexDelete1LogBank => FlexBank {
  Account { },
  Bank { },
  Controller,
  AccountScanner
}
```

Listing 13: Additional reconfiguration edge `FlexDelete1LogBank=>FlexBank` introduced by evolution

  - Drop the configuration {Basic,Flex,Delete,Log} (which is the only configuration that includes the feature Delete).
- Modify the dynamic reconfiguration graph from Figure 5 to create the one depicted in Figure 6 by performing the following operations:
  - Use the feature name Delete1 instead of Delete–also in the names of the nodes. Note that this transformation preserves the currently running product.
  - Add the edge `FlexDelete1LogBank=>FlexBank` as described in Listing 13.

Moreover, as soon as the currently running product reaches a configuration that does not include the feature Delete1, both the product line declaration (in Listing 14) and the dynamic reconfiguration graph of the final Bank PL can be changed by renaming the feature name Delete1 to Delete. This renaming constituted a refactoring and, thus, can be performed safely even if there are pending reconfigurations involving a node that includes the Delete1 feature.

```
features Basic, Flex, Delete1, Log, Corporate
configurations Basic & (Log -> Delete1)
deltas
  { DBasic }
  { DCheck }
  { DFlex when Flex,
    DDelete1 when Delete1 }
  { DLog when Log }
```

Listing 14: Declaration of the final Bank PL

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

Fig. 7 Syntax of classes in IMPERATIVE FEATHERWEIGHT JAVA (IFJ).

3 A Quick Recapitulation of Imperative Featherweight Delta Java (IFΔJ)

In this section, we recall IFΔJ (IMPERATIVE FEATHERWEIGHT DELTA JAVA) (Bettini et al. [2013b]), a core calculus for DOP of product lines of JAVA programs.

3.1 Program Logic with Imperative Featherweight Java (IFJ)

IMPERATIVE FEATHERWEIGHT JAVA (IFJ) is an imperative version of FEATHERWEIGHT JAVA (FJ) (Igarashi et al. [2001]), which supports a more flexible initialization of fields (by field assignment expressions). Within this paper, IFJ serves as core calculus for JAVA to implement the program logic of single products. Both FJ (Igarashi et al. [2001]) and the implementation of IFJ (Bettini et al. [2013b]) define a cast construct. In favor of a more concise presentation of our formalisation, without loss of generality, we do not treat the cast construct explicitly.

The abstract syntax of the IFJ constructs is given in Figure 7. Following Igarashi et al. [2001], we use the overline notation for possibly empty sequences. For instance, we write "$\overline{\texttt{e}}$" as short for a possibly empty sequence of expressions "$\texttt{e}_1, \ldots, \texttt{e}_n$" and "$\overline{\texttt{MD}}$" as short for a possibly empty sequence of method definitions "$\texttt{MD}_1 \ldots \texttt{MD}_n$". The empty sequence is denoted by $\bullet$. We abbreviate operations on sequences of pairs in a similar way, e.g., we write "$\overline{\texttt{C}}\ \overline{\texttt{f}}$" as short for "$\texttt{C}_1\ \texttt{f}_1, \ldots, \texttt{C}_n\ \texttt{f}_n$" and "$\overline{\texttt{C}}\ \overline{\texttt{f}};$" as short for "$\texttt{C}_1\ \texttt{f}_1; \ldots \texttt{C}_n\ \texttt{f}_n;$". Sequences of named elements (field, method or parameter names, field, method or class definitions, ...) are assumed to contain no duplicate names. The set of variables includes the special variable `this` (implicitly bound in any method declaration), which cannot be used as the name of a method's formal parameter.

A class definition class C extends D { $\overline{\texttt{FD}}$; $\overline{\texttt{MD}}$ } consists of its name C, its superclass D (which must always be specified, even if it is Object), a list of field definitions $\overline{\texttt{FD}}$ and a list of method definitions $\overline{\texttt{MD}}$. The fields declared in C are added to the ones declared by D and its superclasses. All fields are assumed to have distinct names (i.e., there is no field shadowing) and public visibility. Each class is assumed to have an implicit constructor that initializes all instance variables to null.

A class table CT is a mapping from class names to class definitions. The subtyping relation $<:$ on classes (types) is the reflexive and transitive closure of the immediate extends

`DM ::= delta` δ `{` $\overline{\texttt{CO}}$ `}`	delta modules
`CO ::= adds CD` | `removes C` | `modifies C [extending C] {` $\overline{\texttt{AO}}$ `}`	class operations
`AO ::= adds FD` | `adds MD` | `removes a` | `modifies MD`	attribute operations

Fig. 8 Syntax of delta modules in IMPERATIVE FEATHERWEIGHT DELTA JAVA (IFΔJ).

relation (the immediate subclass relation, given by the `extends` clauses in `CT`). The class `Object` has no members and its definition does not appear in `CT`. We assume that a class table `CT` satisfies the following sanity conditions: (*i*) $\texttt{CT}(\texttt{C}) = \texttt{class C} \ldots$ for every $\texttt{C} \in \mathit{dom}(\texttt{CT})$ (*ii*) for every class name `C` (except `Object`) appearing anywhere in `CT`, we have $\texttt{C} \in \mathit{dom}(\texttt{CT})$; (*iii*) there are no cycles in the transitive closure of the immediate `extends` relation.

A program is a class table `CT` with a class `class Main { C main() { return(e);} }` for some `C` and `e`.

3.2 Variability with Imperative Featherweight Delta Java (IFΔJ)

IMPERATIVE FEATHERWEIGHT DELTA JAVA (IFΔJ) is an extension to IFJ to support product line development with DOP. The abstract syntax of the IFΔJ constructs is given in Figure 8. The constructs for class definitions `CD`, field definitions `FD` and method definitions `MD` are those of IFJ as presented in Figure 7. Delta module names are denoted by δ. A delta module `DM` (see Figure 8) specifies a sequence of *class operations*. A class operation `CO` can add, remove or modify a class. A class-modify operation possibly specifies the change of the super class and specifies a sequence of *attribute operations*. An attribute operation `AO` can add/remove a field/method or modify a method. A method-modify operation can either replace the method body by another implementation or wrap the existing method using the `original` construct. In both cases, the modified method must have the same signature as the unmodified method.

With the notion of delta modules over IFJ, IFΔJ product lines may be formalized. Feature names are denoted by φ and ψ. Occasionally we use $\overline{\psi}$ to denote also the set of features occurring in the sequence $\overline{\psi}$. A *delta module table* `DMT` is a mapping from delta module names to delta modules. An IFΔJ SPL is a 5-tuple $\texttt{L} = (\overline{\varphi}, \Phi, \texttt{DMT}, \Delta, \Pi)$ consisting of: (*i*) the features $\overline{\varphi}$ of the SPL; (*ii*) the set of the valid feature configurations $\Phi \subseteq \mathscr{P}(\overline{\varphi})$;[5] (*iii*) a delta module table `DMT` containing the delta modules; (*iv*) a mapping $\Delta : \Phi \to \mathscr{P}(\mathit{dom}(\texttt{DMT}))$ determining for which feature configurations a delta module must be applied (which is denoted by the `when` clause in the concrete examples); and (*v*) a totally ordered partition Π of $\mathit{dom}(\texttt{DMT})$, determining the order of delta module application. The 4-tuple $(\overline{\varphi}, \Phi, \Delta, \Pi)$ represents the *product-line declaration*, while the delta module table `DMT` represents the *code base*.

We write $\texttt{CT}_{\overline{\psi}}$ to denote the class table generated for the feature configuration $\overline{\psi}$ and write $<:_{\overline{\psi}}$ to denote the subtype relation associated with the class table $\texttt{CT}_{\overline{\psi}}$. We further write $\mathit{fields}_{\overline{\psi}}(\texttt{C})$ to denote all the fields $\overline{\texttt{FD}}$ of class `C`; $\mathit{meth}_{\overline{\psi}}(\texttt{m}, \texttt{C})$ to denote the definition `MD` of method `m` of class `C`; and $\mathit{subclasses}_{\overline{\psi}}(\texttt{C})$ to denote the subclasses of `C` in $\texttt{CT}_{\overline{\psi}}$. The fields lookup, method lookup and subclasses lookup functions are defined in Figure 9.

The IFΔJ type system (Bettini et al. [2013b], Damiani and Schaefer [2012], Damiani and Lienhardt [2016]) guarantees that, if an SPL `L` is *well typed*, then all its products are well-typed IFJ programs. Hence, for every feature configuration $\overline{\psi} \in \Phi$, the judgement $\vdash \texttt{CT}_{\overline{\psi}}$ **ok**

[5] The calculus abstracts from the feature model concrete representation.

Fields lookup

$$\mathit{fields}_{\overline{\Psi}}(\texttt{Object}) = \bullet \qquad \frac{\mathit{fields}_{\overline{\Psi}}(\texttt{D}) = \overline{\texttt{D}}\ \overline{\texttt{g}} \qquad \texttt{CST}_{\overline{\Psi}}(\texttt{C}) = \texttt{class C extends D}\ \{\ \overline{\texttt{C}}\ \overline{\texttt{f}};\ \overline{\texttt{MH}}\ \}}{\mathit{fields}_{\overline{\Psi}}(\texttt{C}) = \overline{\texttt{D}}\ \overline{\texttt{g}}, \overline{\texttt{C}}\ \overline{\texttt{f}}}$$

Method lookup

$$\mathit{meth}_{\overline{\Psi}}(\texttt{m}, \texttt{C}) = \begin{cases} \texttt{MD} & \text{if } \texttt{MD} = \cdots \texttt{m} \cdots \in \texttt{CT}_{\overline{\Psi}}(\texttt{C}) \\ \mathit{meth}_{\overline{\Psi}}(\texttt{m}, \texttt{D}) & \text{if } \cdots \texttt{m} \cdots \notin \texttt{CT}_{\overline{\Psi}}(\texttt{C}) \text{ and } \texttt{CT}_{\overline{\Psi}}(\texttt{C}) = \texttt{class C extends D}\{\cdots\} \end{cases}$$

Subclasses lookup

$$\mathit{subclasses}_{\overline{\Psi}}(\texttt{C}) = \{\texttt{D} \in \mathit{dom}(\texttt{CT}_{\overline{\Psi}}) \mid \texttt{D} <:_{\overline{\Psi}} \texttt{C}\}$$

Fig. 9 Auxiliary lookup functions for IMPERATIVE FEATHERWEIGHT DELTA JAVA (IFΔJ).

can be derived by the typing rules given in Figure 10. Every well-typed IFJ program is literally a well-typed JAVA program.

4 Runtime Variability with Imperative Featherweight Dynamic Delta Java (IFDΔJ)

IMPERATIVE FEATHERWEIGHT DYNAMIC DELTA JAVA (IFDΔJ) is an extension of IFΔJ for modeling dynamic software product lines (DSPLs) (Hallsteinsen et al. [2008], Capilla et al. [2014]). In this section, we introduce syntax, type system, operational semantics and type soundness of IFDΔJ.

An IFDΔJ DSPL is a 6-tuple $\texttt{L} = (\overline{\varphi}, \Phi, \texttt{DMT}, \Delta, \Pi, \texttt{RG})$ consisting of an IFΔJ SPL $\texttt{L}_0 = (\overline{\varphi}, \Phi, \texttt{DMT}, \Delta, \Pi)$ and a dynamic reconfiguration graph RG. The dynamic product line L is *well typed* if $\texttt{L}_0$ is well typed (cf. end of Section 3) and RG is well typed (cf. the typing rules presented later in this section).

4.1 Syntax of Dynamic Reconfiguration Graphs

A *reconfiguration declaration* R (see Figure 11) consists of:

- an *adjacency declaration* $\overline{\Psi} \Rightarrow \overline{\Psi}'$ specifying that the configuration $\overline{\Psi}$ is adjacent to the configuration $\overline{\Psi}'$ in the reconfiguration graph; and
- a set of *object reconfigurations* $\overline{\texttt{OR}}$ where each object reconfiguration OR specifies how to transform each object of class C in configuration $\overline{\Psi}$ into an object of class C′ in configuration $\overline{\Psi}'$. For each class C, we assume the existence of another default constructor $\texttt{C}(\overline{\texttt{C}}\ \overline{\texttt{f}})\ \{\overline{\texttt{this.f} = \texttt{f};}\}$ that takes initial values for all the fields of C. This constructor can only be invoked in post-reconfiguration expressions.

A dynamic reconfiguration graph RG is a set of reconfiguration declarations with no duplicated adjacency declarations.

Given a reconfiguration declaration $\texttt{R} = \overline{\Psi} \Rightarrow \overline{\Psi}'\ \{\overline{\texttt{OR}}\}$, a class $\texttt{C} \in \mathit{dom}(\texttt{CT}_{\overline{\Psi}})$ is:

- *removed* by R if $\texttt{C} \notin \mathit{dom}(\texttt{CT}_{\overline{\Psi}'})$; and
- *state-modified* by R if it is not removed and $\mathit{fields}_{\overline{\Psi}'}(\texttt{C}) \neq \mathit{fields}_{\overline{\Psi}}(\texttt{C})$.

A reconfiguration declaration must contain an object reconfiguration for each removed or state-modified class and may contain object reconfigurations for non state-modified classes.

Expression typing:

$$\text{(T-VAR)}\ \ \Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x}) \qquad \text{(T-NULL)}\ \ \Gamma \vdash \mathtt{null} : \bot$$

$$\text{(T-NEW)}\ \ \frac{\mathtt{C} \in \mathit{dom}(\mathtt{CT}_{\overline{\Psi}})}{\Gamma \vdash \mathtt{new\ C}() : \mathtt{C}} \qquad \text{(T-FIELD)}\ \ \frac{\Gamma \vdash \mathtt{e} : \mathtt{C} \qquad \mathtt{A\ f} \in \mathit{fields}_{\overline{\Psi}}(\mathtt{C})}{\Gamma \vdash \mathtt{e.f} : \mathtt{A}}$$

$$\text{(T-INVK)}\ \ \frac{\Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0 \qquad \mathit{meth}_{\overline{\Psi}}(\mathtt{m}, \mathtt{C}_0) = \mathtt{B\ m}(\mathtt{A}_1\ _, \ldots, \mathtt{A}_n\ _)\{_\} \qquad \Gamma \vdash \mathtt{e}_i : \mathtt{T}_i\ ^{(i \in 1..n)} \qquad \mathtt{T}_i <:_{\overline{\Psi}} \mathtt{A}_i\ ^{(i \in 1..n)}}{\Gamma \vdash \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) : \mathtt{B}}$$

$$\text{(T-ASSIGN)}\ \ \frac{\Gamma \vdash \mathtt{e}_0.\mathtt{f} : \mathtt{C} \qquad \Gamma \vdash \mathtt{e}_1 : \mathtt{T} \qquad \mathtt{T} <:_{\overline{\Psi}} \mathtt{C}}{\Gamma \vdash \mathtt{e}_0.\mathtt{f} = \mathtt{e}_1 : \mathtt{C}}$$

Method definition typing:

$$\text{(T-METHOD)}\ \ \frac{\mathtt{this} : \mathtt{C}, \overline{\mathtt{x}} : \bar{\mathtt{A}} \vdash \mathtt{e} : \mathtt{T} \qquad \mathtt{T} <:_{\overline{\Psi}} \mathtt{B}}{\mathtt{this} : \mathtt{C} \vdash \mathtt{B\ m}\ (\bar{\mathtt{A}}\ \overline{\mathtt{x}})\{\mathtt{return\ e};\}\ \mathbf{ok}}$$

Class definition typing:

$$\text{(T-CLASS)}\ \ \frac{\mathtt{this} : \mathtt{C} \vdash \overline{\mathtt{MD}}\ \mathbf{ok}}{\vdash \mathtt{class\ C\ extends\ D}\ \{\ \overline{\mathtt{FD}};\ \overline{\mathtt{MD}}\ \}\ \mathbf{ok}}$$

Program typing:

$$\text{(T-PROGRAM)}\ \ \frac{\forall \mathtt{C} \in \mathit{dom}(\mathtt{CT}_{\overline{\Psi}}) \qquad \vdash \mathtt{CT}_{\overline{\Psi}}(\mathtt{C})\ \mathbf{ok}}{\vdash \mathtt{CT}_{\overline{\Psi}}\ \mathbf{ok}}$$

Fig. 10 Typing rules for expressions, methods, classes and the program $\mathtt{CT}_{\overline{\Psi}}$ in IMPERATIVE FEATHERWEIGHT JAVA (IFJ). To reduce clutter, hereafter _ stands for an irrelevant sub-term.

A reconfiguration declaration specifies, given a heap (called the current heap), how to produce a new heap (called the reconfigured heap)—the operation semantics (given in Section 4.3) performs the specified transformation lazily (i.e., each object is reconfigured only when the running product accesses it). An object reconfiguration can be understood as an operation that, given an object of the current heap, introduces a new object with the same address in the reconfigured heap. Given an object o of class C, the reconfiguration operation R behaves as follows:

- If there is an object reconfiguration $\mathtt{OR} = \mathtt{C} \rightarrow \mathtt{C}'\ \{\cdots\}$, then it adds to the reconfigured heap an object $\mathtt{o}'$ of type $\mathtt{C}'$ with the same address of o and initializes *all* its fields $\overline{\mathtt{f}}$ as specified by the instruction in the body of OR.
- If there is no object reconfiguration $\mathtt{OR} = \mathtt{C} \rightarrow \cdots\ \{\cdots\}$, then it copies the object into the reconfigured heap.

A class $\mathtt{C} \in \mathit{dom}(\mathtt{CT}_{\overline{\Psi}})$ is:

- *recoded* by R if, for some superclass $\mathtt{C}_0$ of C in $\overline{\Psi}$ (possibly C itself), either $\mathtt{C}_0$ is removed or $\mathtt{CT}_{\overline{\Psi}'}(\mathtt{C}_0) \neq \mathtt{CT}_{\overline{\Psi}}(\mathtt{C}_0)$;
- *reallocated* by R if R contains an object reconfiguration for C; and

$$
\begin{array}{rll}
\mathtt{R} & ::= & \overline{\psi} \Rightarrow \overline{\psi}' \ \{\overline{\mathtt{OR}}\} & \text{reconfiguration declarations} \\
\mathtt{OR} & ::= & \mathtt{C} \to \mathtt{C}' \ \{\mathtt{pre:}\ \overline{\mathtt{A\,y} = \mathtt{p}};\ \mathtt{post:}\ \overline{\mathtt{B\,z} = \mathtt{q}};\ \overline{\mathtt{this.f} = \mathtt{z}};\} & \text{object reconfigurations} \\
\mathtt{p} & ::= & \mathtt{this} \mid \mathtt{p.f} & \text{pre-reconfiguration expressions} \\
\mathtt{q} & ::= & \mathtt{y} \mid \mathtt{null} \mid \mathtt{new\ C}(\overline{\mathtt{z}}) & \text{post-reconfiguration expressions}
\end{array}
$$

Fig. 11 Syntax of reconfiguration declarations for IMPERATIVE FEATHERWEIGHT DYNAMIC DELTA JAVA (IFDΔJ).

– *affected* by R if it is recoded or reallocated.

We write *affected*(R), *reallocated*(R), *recoded*(R) to denote the set of the names of the classes that are affected, reallocated or recoded by R, respectively. We write $\mathtt{R}(\mathtt{C})$ for the name of the class in which the objects of C are reconfigured by R with the understanding that $\mathtt{R}(\mathtt{C}) = \mathtt{C}$ if $\mathtt{C} \notin$ *reallocated*(R). More precisely:

$$
\mathtt{R}(\mathtt{C}) = \begin{cases} \mathtt{C}' & \text{if } \mathtt{R} = \overline{\psi} \Rightarrow \overline{\psi}' \ \{\ldots, \mathtt{C} \to \mathtt{C}'\{\ldots\}, \ldots\} \\ \mathtt{C} & \text{otherwise} \end{cases}
$$

4.2 Typing Dynamic Reconfiguration Graphs

A reconfiguration graph is well typed if each of its reconfiguration declarations is well typed. The typing rules are listed in Figure 12.

The typing rule for a reconfiguration declaration $\mathtt{R} = \overline{\psi} \Rightarrow \overline{\psi}' \ \{\overline{\mathtt{OR}}\}$ provides guarantees on how objects are reconfigured. Namely, it requires each of its object reconfigurations $\overline{\mathtt{OR}}$ to be well typed and the following condition to be satisfied:

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

The condition prevents a field of class $\mathtt{C}_0$ (that, before the reconfiguration contains the address of an object of some proper subclass C of $\mathtt{C}_0$) from containing, after the reconfiguration, an object whose class ($\mathtt{R}(\mathtt{C})$) is not a subclass of $\mathtt{C}_0$.

The typing rule for object reconfiguration $\mathtt{C} \to \mathtt{C}' \ \{\mathtt{pre:}\ \overline{\mathtt{A\,y} = \mathtt{p}};\ \mathtt{post:}\ \overline{\mathtt{B\,z} = \mathtt{q}};\ \overline{\mathtt{this.f} = \mathtt{z}'};\}$ requires the following condition to be satisfied:

Object reconfiguration condition 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 reconfigured to be instances of a class D′ that is a subclass of B (in $\overline{\psi}'$).

The condition ensures that fields of reconfigured objects are initialized with values of the right type. The typing rule for object reconfiguration uses three different kinds of judgment for typing the pre-reconfiguration assignments ($\overline{\mathtt{A\,y} = \mathtt{p}}$), the post-reconfiguration assignments ($\overline{\mathtt{B\,z} = \mathtt{q}}$) and the fields initializations ($\overline{\mathtt{this.f} = \mathtt{z}'}$), respectively (recall that field initializations involve all the fields of the reconfigured object). These typing rules use the auxiliary lookup functions $\mathit{fields}_{\overline{\psi}}$ and $\mathit{subclasses}_{\overline{\psi}}$ given in Figure 9. The rule for the first kind of judgment (T-PREASSIGN) and the rule for the third kind of judgment (T-POSTINITIALIZEFIELD) perform standard checks within the source feature configuration $\overline{\psi}$ and the target feature configuration $\overline{\psi}'$, respectively. There are three rules for the second kind of judgment, corresponding to the three different kinds of right-hand sides of the assignment. Rule (T-POSTASSIGNNULL) has nothing to check. Rule (T-POSTASSIGNNEW) performs standard checks within feature configuration $\overline{\psi}'$. The most interesting rule is (T-POSTASSIGNVAR), which checks the second object reconfiguration condition illustrated above.

Reconfiguration declaration $\boxed{\vdash \mathtt{R}\ \mathbf{ok}}$

(T-RECONFIGURATION)

$$\frac{\mathtt{R} = \overline{\Psi} \Rightarrow \overline{\Psi}'\ \{\overline{\mathtt{OR}}\} \qquad \mathtt{R} \vdash \mathtt{OR}_i\ \mathbf{ok}^{\ (i=1..|\overline{\mathtt{OR}}|)} \qquad \mathtt{C} <:_{\overline{\Psi}} \mathtt{C}_0\ \textbf{ implies }\ \mathtt{R}(\mathtt{C}) <:_{\overline{\Psi}'} \mathtt{C}_0^{\ (\mathtt{C} \in dom(\mathtt{CT}_{\overline{\Psi}}),\, \mathtt{C}_0 \in dom(\mathtt{CT}_{\overline{\Psi}}) \cap dom(\mathtt{CT}_{\overline{\Psi}'}))}}{\vdash \mathtt{R}\ \mathbf{ok}}$$

Object reconfiguration $\boxed{\mathtt{R} \vdash \mathtt{OR}\ \mathbf{ok}}$

(T-OBJRECONFIGURATION)

$$\frac{\mathtt{R} = \overline{\Psi} \Rightarrow \overline{\Psi}'\ \{\overline{\mathtt{OR}}\} \qquad \overline{\Psi};\mathtt{C} \vdash \mathtt{A}_i\,\mathtt{y}_i = \mathtt{p}_i\ \mathbf{ok}^{\ (i \in 1..|\overline{\mathtt{Ay=p}}|)} \qquad \mathtt{R};\overline{\mathtt{y:A}};\mathtt{z}_1:\mathtt{B}_1,\ldots,\mathtt{z}_{i-1}:\mathtt{B}_{i-1} \vdash \mathtt{B}_i\,\mathtt{z}_i = \mathtt{q}_i\ \mathbf{ok}^{\ (i \in 1..|\overline{\mathtt{Bz=q}}|)} \qquad \overline{\Psi}';\mathtt{C}';\overline{\mathtt{z:B}} \vdash \mathtt{this.f}_i = \mathtt{z}_i\ \mathbf{ok}^{\ (i \in 1..|\overline{\mathtt{this.f=z}}|)}}{\mathtt{R} \vdash \mathtt{C} \to \mathtt{C}'\ \{\mathtt{pre:}\ \overline{\mathtt{Ay=p}};\mathtt{post:}\ \overline{\mathtt{Bz=q}};\ \overline{\mathtt{this.f=z}};\}\ \mathbf{ok}}$$

Pre-reconfiguration assignment $\boxed{\overline{\Psi};\mathtt{C} \vdash \mathtt{Ay=p}\ \mathbf{ok}}$

(T-PREASSIGN)

$$\frac{\overline{\Psi};\mathtt{C} \vdash \mathtt{p} : \mathtt{A}' \qquad \mathtt{A}' <:_{\overline{\Psi}} \mathtt{A}}{\overline{\Psi};\mathtt{C} \vdash \mathtt{Ay=p}\ \mathbf{ok}}$$

Pre-reconfiguration expression $\boxed{\overline{\Psi};\mathtt{C} \vdash \mathtt{p} : \mathtt{D}}$

(T-PREEXPTHIS)

$$_;\mathtt{C} \vdash \mathtt{this} : \mathtt{C}$$

(T-PREEXPFIELD)

$$\frac{\overline{\Psi};\mathtt{C} \vdash \mathtt{p} : \mathtt{D}_0 \qquad \mathtt{Df} \in \mathit{fields}_{\overline{\Psi}}(\mathtt{D}_0)}{\overline{\Psi};\mathtt{C} \vdash \mathtt{p.f} : \mathtt{D}}$$

Post-reconfiguration assignment $\boxed{\mathtt{R};\overline{\mathtt{y:A}};\overline{\mathtt{z:B}} \vdash \mathtt{Bz=q}\ \mathbf{ok}}$

(T-POSTASSIGNVAR)

$$\frac{\mathtt{R} = \overline{\Psi} \Rightarrow \overline{\Psi}'\{_\} \qquad \mathtt{R}(\mathtt{D}) <:_{\overline{\Psi}'} \mathtt{B}^{\ (\mathtt{D} \in \mathit{subclasses}_{\overline{\Psi}}(\mathtt{A}))}}{\mathtt{R};_,\mathtt{y}:\mathtt{A},_;_ \vdash \mathtt{Bz=y}\ \mathbf{ok}}$$

(T-POSTASSIGNNEW)

$$\frac{\mathtt{C} <:_{\overline{\Psi}'} \mathtt{B} \qquad \mathtt{D}_1\,\mathtt{f}_1,\ldots,\mathtt{D}_n\,\mathtt{f}_n = \mathit{fields}_{\overline{\Psi}'}(\mathtt{C}) \qquad \mathtt{z}_i : \mathtt{B}_i \in \overline{\mathtt{z:B}}^{\ (i \in 1..n)} \qquad \mathtt{B}_i <:_{\overline{\Psi}'} \mathtt{D}_i^{\ (i \in 1..n)}}{_ \Rightarrow \overline{\Psi}'\{_\};_;\overline{\mathtt{z:B}} \vdash \mathtt{Bz} = \mathtt{new\ C}(\mathtt{z}_1,\ldots,\mathtt{z}_n)\ \mathbf{ok}}$$

(T-POSTASSIGNNULL)

$$_;_;_ \vdash \mathtt{Bz} = \mathtt{null}\ \mathbf{ok}$$

Post-reconfiguration field initialization $\boxed{\overline{\Psi}';\mathtt{C}';\overline{\mathtt{z:B}} \vdash \mathtt{this.f=z}\ \mathbf{ok}}$

(T-POSTINITIALIZEFIELD)

$$\frac{\mathtt{Df} \in \mathit{fields}_{\overline{\Psi}'}(\mathtt{C}') \qquad \mathtt{B} <:_{\overline{\Psi}'} \mathtt{D}}{\overline{\Psi}';\mathtt{C}';_,\mathtt{z}:\mathtt{B},_ \vdash \mathtt{this.f=z}\ \mathbf{ok}}$$

Fig. 12 Typing rules for reconfiguration declarations for IMPERATIVE FEATHERWEIGHT DYNAMIC DELTA JAVA (IFDΔJ).

4.3 Operational Semantics of Imperative Featherweight Dynamic Delta Java (IFDΔJ)

In order to properly model imperative features, we introduce the concepts of address, value, object, stack and heap. *Addresses*, denoted by ι, are the elements of the denumerable set $\mathbf{I}$. *Values*, denoted by $\mathtt{v}$, are either addresses or `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 $\emptyset$.

Runtime expressions are obtained from expressions (cf. Figure 7) by adding the clause for the expression that models the return from a method call ($\mathtt{return}(\mathtt{e})$) and by replacing all the variables (including `this`) by addresses. We use e to denote runtime expressions.

We introduce *lazy heaps* to account for the lazy reconfiguration of the heap. Lazy heaps are defined by the grammar

$$\mathscr{L} \;::=\; \mathscr{H} \;\mid\; \mathscr{H} : \mathtt{R}(\mathscr{L})$$

Intuitively, 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 the 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; and
- 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 $\mathtt{R}_{i+1}$; and
- $\mathscr{H}_0$ is the heap before the invocation of $\mathtt{R}_1$.

If $dom(\mathscr{H}_1) \supseteq dom(\mathscr{H}_0)$, then all the objects in $\mathscr{H}_0$ have been reconfigured by $\mathtt{R}_1$ into $\mathscr{H}_1$. Therefore, the objects in $\mathscr{H}_0$ are no longer needed and can be garbage collected. In our formalization this amounts to replacing $\mathscr{H}_1 : \mathtt{R}_1(\mathscr{H}_0)$ with $\mathscr{H}_1$. However, the operational semantics does not explicitly model this replacement for simplicity.

We write $\mathscr{L}(\iota) = \langle \mathtt{C}, \overline{\mathtt{f}} = \overline{\mathtt{v}} \rangle$ to mean that $\mathscr{L}$ has either the form $\mathscr{H}$ or $\mathscr{H} : _$ for some $\mathscr{H}$ such that $\mathscr{H}(\iota) = \langle \mathtt{C}, \overline{\mathtt{f}} = \overline{\mathtt{v}} \rangle$. We further write $\mathscr{L} \cup \{\iota \mapsto \cdots\}$ to denote the lazy heap obtained from $\mathscr{L}$ where the association $\iota \mapsto \cdots$ in $\mathscr{H}$ has been added. Similarly, we write $\mathscr{L}[\iota \mapsto \cdots]$ to denote the lazy heap obtained from $\mathscr{L}$ where the association $\iota \mapsto \cdots$ in $\mathscr{H}$ has been updated.

4.3.1 Reduction Rules

The *state* of a computation is a 4-tuple $(\overline{\psi}, \mathscr{L}, \overline{\iota}, e)$ consisting of a current feature configuration $\overline{\psi}$, a lazy heap $\mathscr{L}$, a stack $\overline{\iota}$ recording the addresses of the objects on which the running methods have been invoked and a runtime expression e representing the bodies of the active methods.

States evolve according to the relation $\Longrightarrow$, defined in Figure 13, by exploiting the auxiliary relation $\longrightarrow$ which describes the computations within a given feature configuration. The computation rules use the auxiliary lookup functions $\mathit{fields}_{\overline{\psi}}$ and $\mathit{subclasses}_{\overline{\psi}}$ given in Figure 9. The relation $\Longrightarrow$ extends $\longrightarrow$ by enabling reconfigurations whereby the features $\overline{\psi}$ can be changed into $\overline{\psi}'$ if $\overline{\psi}$ and $\overline{\psi}'$ are adjacent. The reconfiguration rule can be applied nondeterministically whenever the current state of the computation satisfies the predicate $\mathit{Enabled}(\mathtt{R}, \mathscr{L}, \overline{\iota}, e)$, which expresses whether the reconfiguration $\mathtt{R}$ is enabled. We will

come back to the *Enabled* predicate later. Note that computation and congruence rules never change the feature configuration (to avoid clutter, in the rules we replace the component $\overline{\psi}$ with _).

The relation $\longrightarrow$ is defined in terms of fairly standard notions of *computation rules* and *congruence rules*. It ensures that the computation is carried out according to a call-by-value reduction strategy. Congruence rules are standard. The only non-standard feature of $\longrightarrow$, which is in fact specific to IFDΔJ, is the use of the auxiliary function *olookup* for accessing objects in the lazy heap. We will come back to *olookup* shortly, when we describe the lazy heap reconfiguration mechanism.

The *initial program state* associated with a program $\mathtt{CT}_{\overline{\psi}}$ is $(\overline{\psi}, \emptyset, \bullet, \mathtt{e})$ where $\mathtt{return(e)}$ is the body of the `Main` method, that is $\mathtt{CT}_{\overline{\psi}}(\mathtt{Main}) = \mathtt{class\ Main\ \{\ C\ main()\ \{\ return(e);\}\ \}}$. We write $\mathtt{C}_{\mathtt{Main}}$ to denote the return type `C` of the `main` method of class `Main`.

4.3.2 Object Lookup and Lazy Heap Reconfiguration

The mutually recursive auxiliary functions *olookup*, *oreconf* and *preeval* are defined in Figure 14. Intuitively, the object lookup function *olookup* returns the object at address ι in the lazy heap $\mathscr{L}$, along with a possibly updated lazy heap $\mathscr{L}'$. In the simplest cases, corresponding to the first two rules defining *olookup*, the object at ι is already up-to-date with respect to the most recent reconfiguration (or no reconfiguration has occurred yet) so that *olookup* returns the object in its present state along with an unchanged lazy heap. If the object is not present in the most recently reconfigured heap ($\iota \notin dom(\mathscr{H})$), then the object is first looked up recursively in the lazy heap $\mathscr{L}$ that immediately precedes the last reconfiguration `R` and then it is reconfigured by means of the *oreconf* auxiliary function.

The purpose of $oreconf(\iota, \mathtt{R}, \mathscr{L})$ is to reconfigure the object located at ι (which is assumed to be found in the topmost heap of $\mathscr{L}$) according to `R`. The function returns a triple consisting of the reconfigured object, a heap of new objects that have been created as a consequence of the reconfiguration of ι and the new lazy heap. There are two rules defining *oreconf*: the first one deals with implicitly reconfigured objects (those whose class `C` has no reconfiguration clause in `R`), which are returned unchanged; the second rule deals with objects whose class is explicitly reallocated by a clause $\mathtt{C} \to \mathtt{C}'\ \{\mathtt{pre:}\ \overline{\mathtt{A\,y = p}};\ \mathtt{post:}\ \overline{\mathtt{B\,z = q}};\ \overline{\mathtt{this.f = z'}};\}$, which specifies the new class $\mathtt{C}'$ of the object and the rearrangements of its fields in the new configuration. The pre-reconfiguration assignments $\overline{\mathtt{A\,y = p}}$ retrieve further objects necessary for the reconfiguration by means of the *preeval* function. Notice that *preeval* may trigger further reconfigurations (it calls *olookup* recursively) and that it defaults to `null` any attempt at accessing `null` addresses (this is to ensure progress during the reconfiguration phase).[6]

The post-reconfiguration assignments $\overline{\mathtt{B\,z = q}}$ take care of the proper migration from the old to new feature configuration of classes. In particular, pointers to either new or existing (but possibly not reconfigured) objects `q` are assigned to variables `z` whose types refer to the new configuration. Finally, the assignments $\overline{\mathtt{this.f = z'}}$ initialize the fields of the newly created object of class $\mathtt{C}'$. Overall, the operational semantics implements a high degree of laziness in the sense that objects are reconfigured only when necessary (either because one of their fields is accessed or because a method is invoked on them).

[6] In a full-fledged language, the access to `null` would raise an exception for which the reconfiguration operation should provide an appropriate handler.

Reduction rules $\boxed{\overline{\psi}, \mathscr{L}, \overline{\iota}, e \Longrightarrow \overline{\psi}', \mathscr{L}', \overline{\iota}', e'}$

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

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

Computation rules $\boxed{\overline{\psi}, \mathscr{L}, \overline{\iota}, e \longrightarrow \overline{\psi}, \mathscr{L}', \overline{\iota}', e'}$

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

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

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

$$\text{(C-INVK)}\quad \frac{\mathit{olookup}(\iota, \mathscr{L}) = \langle \mathtt{C}, \ldots \rangle, \mathscr{L}' \qquad \mathit{meth}_{\overline{\psi}}(\mathtt{m}, \mathtt{C}) = _\ \mathtt{m}(_\ \overline{\mathtt{x}})\{\mathtt{return\ e}_0;\}}{_, \mathscr{L}, \overline{\iota}, \iota.\mathtt{m}(\overline{\mathtt{v}}) \longrightarrow _, \mathscr{L}', \overline{\iota}\iota, \mathtt{return}([\overline{\mathtt{x}}/\overline{\mathtt{v}}, \mathtt{this}/\iota]\mathtt{e}_0)}$$

$$\text{(C-RET)}\quad _, \mathscr{L}, \overline{\iota}\iota, \mathtt{return}(\mathtt{v}) \longrightarrow _, \mathscr{L}, \overline{\iota}, \mathtt{v}$$

Congruence rules

$$\frac{_, \mathscr{L}, \overline{\iota}, e \longrightarrow _, \mathscr{L}', \overline{\iota}', e'}{_, \mathscr{L}, \overline{\iota}, e.\mathtt{f} \longrightarrow _, \mathscr{L}', \overline{\iota}', e'.\mathtt{f}} \qquad \frac{_, \mathscr{L}, \overline{\iota}, e \longrightarrow _, \mathscr{L}', \overline{\iota}', e'}{_, \mathscr{L}, \overline{\iota}, e.\mathtt{m}(\overline{e}) \longrightarrow _, \mathscr{L}', \overline{\iota}', e'.\mathtt{m}(\overline{e})}$$

$$\frac{_, \mathscr{L}, \overline{\iota}, e_i \longrightarrow _, \mathscr{L}', \overline{\iota}', e_i'}{_, \mathscr{L}, \overline{\iota}, \mathtt{v.m}(\overline{\mathtt{v}}, e_i, \overline{e}) \longrightarrow _, \mathscr{L}', \overline{\iota}', \mathtt{v.m}(\overline{\mathtt{v}}, e_i', \overline{e})} \qquad \frac{_, \mathscr{L}, \overline{\iota}, e \longrightarrow _, \mathscr{L}', \overline{\iota}', e'}{_, \mathscr{L}, \overline{\iota}, e.\mathtt{f} = e_0 \longrightarrow _, \mathscr{L}', \overline{\iota}', e'.\mathtt{f} = e_0}$$

$$\frac{_, \mathscr{L}, \overline{\iota}, e \longrightarrow _, \mathscr{L}', \overline{\iota}', e'}{_, \mathscr{L}, \overline{\iota}, \mathtt{v.f} = e \longrightarrow _, \mathscr{L}', \overline{\iota}', \mathtt{v.f} = e'} \qquad \frac{_, \mathscr{L}, \overline{\iota}, e \longrightarrow _, \mathscr{L}', \overline{\iota}', e'}{_, \mathscr{L}, \overline{\iota}, \mathtt{return}(e) \longrightarrow _, \mathscr{L}', \overline{\iota}', \mathtt{return}(e')}$$

Fig. 13 Reduction, computation and congruence rules in IMPERATIVE FEATHERWEIGHT DYNAMIC DELTA JAVA (IFDΔJ).

4.3.3 The Enabled Predicate

The predicate $\mathit{Enabled}(\mathtt{R}, \mathscr{L}, \overline{\iota}, e)$ expresses the runtime checks to be performed in order to ensure that the reconfiguration $\mathtt{R} = \overline{\psi} \Rightarrow \overline{\psi}'\ \{\overline{\mathtt{OR}}\}$ can be safely performed in a given state $(\overline{\psi}, \mathscr{L}, \overline{\iota}, e)$ of the computation. It consists of the two following conditions:

Enabling condition 1 All methods that are currently executing are invoked on objects belonging to classes that are not affected by the reconfiguration; and

Enabling condition 2 The runtime expression e is well typed in configuration $\overline{\psi}'$ and its type is a subtype of the return type of the method `main` of class `Main`.

Object lookup $\boxed{olookup(\iota, \mathscr{L}) = \langle \mathtt{C}, \overline{\mathtt{f}} = \overline{\mathtt{v}} \rangle, \mathscr{L}'}$

$$olookup(\iota, \mathscr{H}) = \mathscr{H}(\iota), \mathscr{H} \qquad \frac{\mathscr{L} = \mathscr{H} : _ \qquad \iota \in dom(\mathscr{H})}{olookup(\iota, \mathscr{L}) = \mathscr{H}(\iota), \mathscr{L}}$$

$$\frac{\iota \notin dom(\mathscr{H}) \qquad olookup(\iota, \mathscr{L}) = _, \mathscr{L}' \qquad oreconf(\iota, \mathtt{R}, \mathscr{L}') = \langle \mathtt{C}, \overline{\mathtt{f}} = \overline{\mathtt{v}} \rangle, \mathscr{H}', \mathscr{L}''}{olookup(\iota, \mathscr{H} : \mathtt{R}(\mathscr{L})) = \langle \mathtt{C}, \overline{\mathtt{f}} = \overline{\mathtt{v}} \rangle, \mathscr{H} \cup \{\iota \mapsto \langle \mathtt{C}, \overline{\mathtt{f}} = \overline{\mathtt{v}} \rangle\} \cup \mathscr{H}' : \mathtt{R}(\mathscr{L}'')}$$

Object reconfiguration $\boxed{oreconf(\iota, \mathtt{R}, \mathscr{L}) = \langle \mathtt{C}, \overline{\mathtt{f}} = \overline{\mathtt{v}} \rangle, \mathscr{H}, \mathscr{L}'}$

$$\frac{\mathscr{L}(\iota) = \langle \mathtt{C}, \overline{\mathtt{f}} = \overline{\mathtt{v}} \rangle \qquad \mathtt{C} \notin reallocated(\mathtt{R})}{oreconf(\iota, \mathtt{R}, \mathscr{L}) = \langle \mathtt{C}, \overline{\mathtt{f}} = \overline{\mathtt{v}} \rangle, \emptyset, \mathscr{L}}$$

$$\frac{\begin{array}{c}\mathtt{R} = \overline{\Psi} \Rightarrow \overline{\Psi}' \{ \cdots \mathtt{C} \to \mathtt{C}' \ \{\mathtt{pre:}\ \overline{\mathtt{A}\,\mathtt{y} = \mathtt{p}};\ \mathtt{post:}\ \overline{\mathtt{B}\,\mathtt{z} = \mathtt{q}};\ \overline{\mathtt{this.f} = \mathtt{z}'};\} \cdots \} \\ \mathscr{L}(\iota) = \langle \mathtt{C}, \ldots \rangle \qquad preeval(\overline{\mathtt{p}}[\iota/\mathtt{this}], \mathscr{L}) = \overline{\mathtt{v}}, \mathscr{L}' \qquad postassign(\overline{\Psi}', \overline{\mathtt{B}\,\mathtt{z} = \mathtt{q}}[\overline{\mathtt{v}}/\overline{\mathtt{y}}]) = \overline{\mathtt{u}}, \mathscr{H}\end{array}}{oreconf(\iota, \mathtt{R}, \mathscr{L}) = \langle \mathtt{C}', \overline{\mathtt{f}} = \overline{\mathtt{z}'}[\overline{\mathtt{u}}/\overline{\mathtt{z}}] \rangle, \mathscr{H}, \mathscr{L}'}$$

Pre-reconfiguration auxiliary function $\boxed{preeval(\overline{\mathtt{p}}, \mathscr{L}) = \overline{\mathtt{v}}, \mathscr{L}'}$

$$preeval(\bullet, \mathscr{L}) = \bullet, \mathscr{L} \qquad \frac{\overline{\mathtt{p}} \neq \bullet \qquad preeval(\mathtt{p}, \mathscr{L}) = \mathtt{v}, \mathscr{L}' \qquad preeval(\overline{\mathtt{p}}, \mathscr{L}') = \overline{\mathtt{v}}, \mathscr{L}''}{preeval(\mathtt{p}\overline{\mathtt{p}}, \mathscr{L}) = \mathtt{v}\overline{\mathtt{v}}, \mathscr{L}''}$$

$$preeval(\iota, \mathscr{L}) = \iota, \mathscr{L} \qquad \frac{preeval(\mathtt{p}, \mathscr{L}) = \mathtt{null}, \mathscr{L}'}{preeval(\mathtt{p.f}, \mathscr{L}) = \mathtt{null}, \mathscr{L}'}$$

$$\frac{preeval(\mathtt{p}, \mathscr{L}) = \iota, \mathscr{L}' \qquad olookup(\iota, \mathscr{L}') = \langle \mathtt{C}, \overline{\mathtt{f}} = \overline{\mathtt{u}} \rangle, \mathscr{L}''}{preeval(\mathtt{p.f}_i, \mathscr{L}) = \mathtt{u}_i, \mathscr{L}''}$$

Post-reconfiguration auxiliary functions $\boxed{postassign(\overline{\Psi}, \overline{\mathtt{B}\,\mathtt{z} = \mathtt{q}}) = \overline{\mathtt{v}}, \mathscr{H}}$ **and** $\boxed{posteval(\overline{\Psi}, \mathtt{q}) = \mathtt{v}, \mathscr{H}}$

$$postassign(\overline{\Psi}, \bullet) = \bullet, \emptyset \qquad \frac{posteval(\overline{\Psi}, \mathtt{q}) = \mathtt{v}, \mathscr{H} \qquad postassign(\overline{\Psi}, \overline{\mathtt{B}\,\mathtt{z} = \mathtt{q}}[\mathtt{v}/\mathtt{z}]) = \overline{\mathtt{v}}, \mathscr{H}'}{postassign(\overline{\Psi}, \mathtt{B}\,\mathtt{z} = \mathtt{q}\overline{\mathtt{B}\,\mathtt{z} = \mathtt{q}}) = \mathtt{v}\overline{\mathtt{v}}, \mathscr{H} \cup \mathscr{H}'}$$

$$posteval(\overline{\Psi}, \mathtt{v}) = \mathtt{v}, \emptyset \qquad \frac{\iota \text{ fresh} \qquad fields_{\overline{\Psi}}(\mathtt{C}) = \overline{\mathtt{C}\ \mathtt{f}}}{posteval(\overline{\Psi}, \mathtt{new\ C}(\overline{\mathtt{v}})) = \iota, \{\iota \mapsto \langle \mathtt{C}, \overline{\mathtt{f}} = \overline{\mathtt{v}} \rangle\}}$$

Fig. 14 Object lookup in a lazy heap for IMPERATIVE FEATHERWEIGHT DYNAMIC DELTA JAVA (IFDΔJ).

The first condition aims to prevent inconsistent behavior. It is formalized as

$$classof(\iota_i, \mathscr{L}) \notin affected(\mathtt{R})^{\ (1 \leq i \leq n)}$$

where $\overline{\iota} = \iota_1 \cdots \iota_n$ is the stack. The function $classof(\iota, \mathscr{L})$ returns the name of the class of the object of address ι in the already reconfigured part of the lazy heap $\mathscr{L}$ and is defined by: $classof(\iota, \mathscr{L}) = \mathtt{C}$, if $\mathscr{L}(\iota) = \langle \mathtt{C}, \ldots \rangle$. As the stack $\overline{\iota}$ contains addresses of objects on which there is an active method invocation, such objects necessarily occur in the already reconfigured part of the lazy heap $\mathscr{L}$.

The second condition ensures that subject reduction (which is used to prove type soundness) holds and is formalized by the judgment

$$\Sigma \vdash_{\overline{\Psi}} e : \mathtt{T}$$

which must hold for some $\mathtt{T} <:_{\overline{\Psi}'} \mathit{meth}_{\overline{\Psi}'}(\mathtt{main},\mathtt{Main})$,[7] where the *heap environment* $\Sigma = \{\iota : \mathit{clookup}(\iota,\mathscr{L}) \mid \iota \text{ occurs in } e\}$ maps each address occurring in e to the name of the class of the object at the address ι in the new configuration $\overline{\Psi}'$. The auxiliary function $\mathit{clookup}(_,_)$ retrieves the class name of an object updated according to all the reconfigurations occurred since its first allocation and is defined as follows

$$\mathit{clookup}(\iota,\mathscr{H}) = \mathtt{C} \qquad \text{if } \mathscr{H}(\iota) = \langle \mathtt{C},\ldots\rangle$$
$$\mathit{clookup}(\iota,\mathscr{H}:\mathtt{R}(\mathscr{L})) = \begin{cases} \mathtt{C} & \text{if } \mathscr{H}(\iota) = \langle \mathtt{C},\ldots\rangle \\ \mathtt{R}(\mathit{clookup}(\iota,\mathscr{L})) & \text{if } \iota \notin \mathit{dom}(\mathscr{H}) \end{cases}$$

The second condition of the predicate $\mathit{Enabled}(\mathtt{R},\mathscr{L},\overline{\iota},e)$ requires type checking the runtime expression e, which is the outcome of reduction steps performed on the bodies of the active methods. In the considered core calculus, this condition is needed because the types (classes) of the subexpressions of a method body do not (necessarily) occur within the method definition. Instead, it could be dropped in a formalization where the type of each subexpression of a method body occurs in the method definition, e.g., where method definitions have the *single static assignment* (SSA) form (Cytron et al. [1991]):

$$\mathtt{C}\,\mathtt{m}(\mathtt{A}_1\,\mathtt{x}_1,\ldots,\mathtt{A}_p\,\mathtt{x}_p)\,\{\mathtt{B}_1\,\mathtt{y}_1 = e_1;\ldots;\mathtt{B}_q\,\mathtt{y}_q = e_q;\ \mathtt{return}\ \mathtt{y}_q\}$$

where (for all $i \in 1..q$) $\mathtt{e}_i$ is of the form either $\mathtt{z}$, or $\mathtt{z.f}$, or $\mathtt{z.f} = \mathtt{z}'$, or $\mathtt{z.m}(\overline{\mathtt{z}})$ for some $\mathtt{z},\mathtt{z}',\overline{\mathtt{z}} \in \{\mathtt{x}_1,\ldots,\mathtt{x}_n,\mathtt{y}_1,\ldots,\mathtt{y}_{i-1}\}$. In fact, in such a formalization, the second condition of the predicate $\mathit{Enabled}(\mathtt{R},\mathscr{L},\overline{\iota},e)$ would be implied by the first object reconfiguration condition (which is statically checked by (T-RECONFIGURATION) of Fig. 12) because, by the first condition of the predicate, all the classes occurring in the definition of an active method are defined both in the current and in the reconfigured program.

4.4 Type Soundness of Imperative Featherweight Dynamic Delta Java (IFDΔJ)

Type soundness is stated as follows (the notion of initial state and the notation $\mathtt{C}_{\mathtt{Main}}$ have been introduced at the end of Section 4.3.1).

Theorem 1 (Type soundness) *Let* $\mathtt{L} = (\overline{\Psi},\Phi,\mathtt{DMT},\Delta,\Pi,\mathtt{RG})$ *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*

1. `null`*, or*
2. *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*
3. *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* $\overline{\mathtt{v}}$*.*

For the sake of simplicity, the theorem is stated for a reduction sequence starting from an initial state in which both the heap and the stack are empty. However, since the statement considers an arbitrarily long sequence of reductions, the resulting state for which properties 1–3 hold may have been reached after an arbitrary number of reconfigurations. The proof of the theorem is done using the standard technique of subject reduction and progress. In particular, the subject reduction theorem considers reduction steps starting from more general configurations of stack and heap. The characterization of those configurations that are reachable from the initial state of a well-typed SPL requires additional definitions. The details can be found in Appendix A.

[7] The first enabling condition ensures that $\mathit{meth}_{\overline{\Psi}'}(\mathtt{main},\mathtt{Main}) = \mathit{meth}_{\overline{\Psi}}(\mathtt{main},\mathtt{Main})$.

5 Related Work

5.1 Feature-Oriented Programming (FOP)

Feature-oriented programming (FOP) (Batory et al. [2004], Kästner et al. [2008], Schaefer et al. [2012]) is a compositional approach for implementing SPLs in which code fragments are associated with product features and assembled to implement a particular configuration of features. Other compositional approaches use aspects (Kästner et al. [2007], Aracic et al. [2006]), mixins (Smaragdakis and Batory [2002]), hyperslices (Tarr et al. [1999]) or traits (Ducasse et al. [2006]) to implement product line variability (see Lopez-Herrejon et al. [2005] as well as Bettini et al. [2013c] for a discussion of some of them with respect to FOP).

For the implementation of dynamic SPLs (Hallsteinsen et al. [2008], Capilla et al. [2014]) in FOP, (Rosenmüller et al. [2011a]) support flexible feature binding to allow selecting features statically at compile-time using superimposition or dynamically at build-time using the decorator pattern. The variability of feature binding is achieved by code transformations for integrating static and dynamic feature bindings. They also use transformation rules on feature models to provide composition safety of dynamic binding. Rosenmüller et al. [2011b] extend their approach to support runtime adaptation and self-configuration on top of flexible binding units. They use feature-based adaptation rules to describe SPL adaptation in a declarative way. The adaptation mechanism transforms the feature model of an SPL according to the binding units of the generated DSPL, thus allowing to change the behavior of a program at run-time. However, this approach for dynamic adaptation does not allow to change the state of the program (i.e., a dynamic reconfiguration cannot add/remove or change the value of fields), which is in contrast to the mechanism proposed in this article.

FeatureC++ (Apel et al. [2005]) provides means to dynamically compose feature modules. In particular, the authors investigate the combination of FOP and aspect-oriented programming (AOP) to eliminate shortcomings of FOP to capture dynamic cross-cutting modularity. However, runtime reconfiguration including an update of the heap structures according to the new feature configuration is not supported. Günther and Sunkle [2010] present an extended version of rbFeatures, a FOP implementation in Ruby, which provides runtime adaptation, variant modification and configuration of software product lines. New features can be added at runtime, but other change operations, such as deselecting or removing features, are not supported. Apel et al. [2010] propose the FFJ/FFJ$_{PL}$ core calculus that models FOP of SPLs of FEATHERWEIGHT JAVA programs. Although in FFJ/FFJ$_{PL}$, feature composition is modeled as a static process (done before compilation), the formalization leaves open when feature composition is performed. Therefore, it could be used to model dynamic feature composition at run time. As DOP is an extension of FOP (Schaefer and Damiani [2010]), the mechanism for runtime adaptation and dynamic evolution of product lines proposed in this article can also be applied to FOP product lines.

5.2 Aspect-Oriented Programming (AOP)

Aspect-oriented programming (AOP) (Kiczales et al. [1997]) has been used to implement SPLs (Groher and Voelter [2009], Alves et al. [2007]) and supports the dynamic selection of aspects at runtime, e.g., in *CeasarJ* (Aracic et al. [2006]) or *AspectJ* (Kiczales et al. [2001]). Aspects do not add or remove existing fields so that heap updates are not necessary when aspects are added or removed at runtime. Thus, dynamic DOP supports more

flexible changes of functionality at runtime. For a detailed comparison of AOP and DOP consider (Bettini et al. [2013c]).

Chakravarthy et al. [2008] describe a technique to provide binding-time flexibility in a modular and convenient manner. They use a combination of design-patterns and AOP to achieve binding-time flexibility. A pattern encapsulates the variation point and targeted aspects set the binding times of the pattern participants. However, they do not consider the evolution of feature models and, therefore, handle only anticipated change. Ribeiro et al. [2009] investigate whether AspectJ provides modularity when implementing features with flexible binding times. This study leads to the conclusion that, in a general case, AspectJ does not provide modularity in a DSPL. Dinkelaker et al. [2010] propose an approach for DSPLs which combines dynamic aspects, runtime models of aspects, as well as detection and resolution of aspect interactions but they do not consider the change of features at runtime. Andrade et al. [2014] create three AspectJ-based idioms to implement flexible feature bindings and evaluate those using case studies. The idioms are based on exploiting specific language features of AspectJ and the aspect weaving capabilities, but they do not provide means to update existing objects.

5.3 General Object-oriented Programming (OOP)

In general object-oriented programming (OOP), there are several approaches to modify the functionality of objects at runtime. Primitives for *dynamic object reclassification* (i.e., for changing at runtime the class membership of an object) are present, e.g., in the dynamically typed languages SMALLTALK and CLOS. In the programming language *gbeta* (Ernst [1999]), classes can be composed dynamically and objects can be reassigned to other classes at runtime. The proposed mechanism is relatively flexible but it is not type safe.

Fickle$_{II}$ (Drossopoulou et al. [2002]) is a core JAVA-like object-oriented language where objects are allowed to change their class at runtime. Ancona et al. [2007] have developed a translation from *Fickle* into JAVA that has proven correctness. In *Fickle*$_{II}$, only objects belonging to special classes, called *root* and *state* classes, can be reclassified and the type system restricts the use of these classes (in particular, state classes may not be used as types for fields). The *Fickle*$_3$ calculus (Damiani et al. [2003]) eliminates the need to declare explicitly the classes of the objects that may be reclassified. Reclassification may be decided by the client of a class, allowing unanticipated object reclassification. However, the type system restricts the use of the classes of the objects that may be re-classified. More recently, *typestate-oriented programming* (Aldrich et al. [2009], Saini et al. [2010]) has overcome some of the limitations in *Fickle*$_{II}$/*Fickle*$_3$, e.g., the inability to track the states of fields.

In Bettini and Bono [2008] as well as Bettini et al. [2011], an extension of FJ with object composition and delegation is presented. In that calculus, methods can be changed dynamically at runtime on an existing object as a consequence of object composition and "redefining" methods (a runtime version of standard method overriding).

Dynamic trait replacement is the ability to change the behavior of an object at runtime by replacing one trait for another. In the prototype-based language SELF, dynamic trait replacement can be achieved by changing the reference to the parent of an object. In the class-based setting, dynamic trait replacement has been formalized through the JAVA-like language *Chai*$_3$ (Smith and Drossopoulou [2005]). The language *Chai*$_3$ contains an operator for replacing a trait in an existing object. This operator requires that the trait to be replaced corresponds exactly to a named trait used in the object's class definition. This makes the

flexibility/expressivity of $Chai_3$ similar to the one of $Fickle_{II}$ (Drossopoulou et al. [2002]). Bettini et al. [2013a] propose a more flexible dynamic trait replacement operator.

In *ObjectTeams* (Herrmann [2007]), objects can be assigned different roles at runtime such that the behavior may change dynamically. *Context-oriented programming* (Hirschfeld et al. [2008]) allows defining several layers of behavior that can be switched on or off dynamically at runtime.

Dynamic classes (Johnsen et al. [2009]) perform (in a type-sound manner) run-time updates of object-oriented systems by adding or refining classes or by removing redundant program parts. In particular, Johnsen et al. [2009] model asynchronous updates in presence of concurrency, while the formalization of dynamic DOP presented in this paper does not model concurrency.

A recently proposed aspect-based dynamic software updating model combines object-oriented and aspect-oriented techniques to build an update analyzer that automatically compares two versions of a JAVA program and extracts the necessary updates expressed as aspects (Cech Previtali and Gross [2011]). JAVADAPTOR (Pukall et al. [2013]) is a dynamic software update approach that provides JAVA with the same runtime capabilities known from dynamically typed languages (possibly causing program inconsistencies) and runs on basis of all major standard JAVA virtual machines. Subsequently, extensions of the JAVA HotSpot VM that support type-safe flexible dynamic software updates have been proposed (Würthinger et al. [2013], Gu et al. [2014]). In particular, Gu et al. [2014] support lazy object updates by means of a *transformer* construct that closely resembles our object reconfiguration construct (although it does not support migration of objects from one class to another with a different name).

While the above approaches could be used to implement dynamic SPLs, neither of them relates the changes in the altered functionalities to product features. Dynamic DOP combines an established software product line engineering approach with the capability of type-sound runtime reconfiguration and runtime evolution.

6 Conclusion and Future Work

We presented a core calculus providing a formal definition of dynamic DOP. DOP is an approach to construct SPLs. Dynamic DOP adds to DOP a dynamic reconfiguration graph which specifies which configurations can be updated at runtime, and how to update the program's heap to go to another configuration. A Bank Product Line example was used to illustrate both DOP and dynamic DOP: the generation of a product was presented, as well as several scenarios of runtime update. The dynamic reconfiguration graph was decoupled from the notion of DOP so that it could be used in connection with other approaches for constructing SPLs. The core calculus is equipped with a type system that was proved to be sound.

With this formal foundation of dynamic delta-oriented programming, it is possible to switch the implemented product configuration at runtime. Furthermore, it is also possible to perform (unanticipated) evolution of the product line declaration, the dynamic reconfiguration graph, and the code base of an SPL with updates as soon as possible while still preserving the currently active product. Finally, the type system of our dynamic DOP core calculus ensures that the dynamic reconfigurations lead to type safe products and do not cause runtime type errors.

The core calculus does not model multi-threaded applications and we do not address implementation issues. The transition between products depends on both the reconfigura-

$$\text{(WF-HeapEnv)}\quad \frac{\Sigma(\iota) \in dom(\mathtt{CT}_{\overline{\Psi}})^{\ (\iota \in dom(\Sigma))}}{\Vdash_{\overline{\Psi}} \Sigma}$$

$$\text{(WF-LazyHeapEnv)}\quad \frac{\Vdash_{\overline{\Psi}} \Sigma \qquad \Vdash_{\overline{\Psi}'} \Theta \qquad \mathtt{R} = \overline{\Psi}' \Rightarrow \overline{\Psi}\,\{_\} \qquad (dom(\Sigma) \setminus head\text{-}dom(\Theta)) \cap tail\text{-}dom(\Theta) = \emptyset \qquad \Sigma(\iota) = \mathtt{R}(\Theta(\iota))^{\ (\iota \in dom(\Sigma) \cap head\text{-}dom(\Theta))}}{\Vdash_{\overline{\Psi}} \Sigma : \mathtt{R}(\Theta)}$$

Fig. 15 IFDΔJ: Rules for well-formed heap environments.

tion graph and the current system state. A limitation of the proposed approach is that, if the method of some classes never leave the call stack (e.g., the main method for simple JAVA programs) changes to these classes, let us call them *fixed* classes, will not be possible with dynamic DOP (without a system shutdown). Therefore, an arbitrary product line can be configured to a product for which the reconfiguration graph will disallow a further product change. This limitation can be mitigated by adopting suitable programming patterns that limit the number of fixed classes and allow to significantly change the behavior of the application without changing these classes. In future work, we aim to extend the calculus to model multi-threaded applications.

Recently, proof systems for the verification of delta-oriented SPLs (Hähnle and Schaefer [2012], Damiani et al. [2012a], Bubel et al. [2016]) and model-based testing frameworks for delta-oriented SPLs (Lochau et al. [2012], Damiani et al. [2013]) have been proposed. In future work, we plan to extend these approaches to dynamic delta-oriented SPLs.

A prototypical implementation of DOP which supports SPLs of JAVA 1.5 programs is available (Koscielny et al. [2014]). In future work, we would like to investigate the possibility to implement dynamic DOP for (a suitable subset of) JAVA. This implementation poses several challenges. In particular: the proposed approach goes beyond the capabilities of a regular JAVA VM (although, as pointed out in Section 5.3, extensions of the JAVA HotSpot VM that support flexible dynamic software updates have been recently proposed (Würthinger et al. [2013], Gu et al. [2014])); the memory footprint of skipping the updated code can be significant; managing multiple heaps would require a customized runtime environment; and interaction with the garbage collection process and multi-threading must be addressed. Another interesting direction for future work is to investigate software engineering methodologies and tools for supporting evolution of dynamic delta-oriented SPL: evolving complex systems with many classes and feature interactions is quite challenging.

A Proof of Theorem 1 (Type Soundness)

In order to be able to formulate the type soundness of IFDΔJ by means of a subject-reduction theorem and a progress theorem for the small-step semantics, we need to formulate a type system for runtime expressions. Expressions containing a *stupid selection*, i.e., a field selection `null.f` or a method invocation `null.m`$(\cdots)$, are not well typed according to the IFJ source level type system (cf. Figure 10). However, a runtime expression without stupid selections may reduce to a runtime expression containing a stupid selection. The type system for runtime expressions contains a rule for assigning any type `T` to the value `null` (so that stupid selection can be typed).

An *heap (type) assumption* Σ is a mapping from *addresses* to *class names*. The empty-heap assumption is denoted by $\bullet$. A *lazy-heap assumption* Θ is either a heap assumption Σ or a partially reconfigured heap assumption of the form

$$\Sigma_n : \mathtt{R}_n(\Sigma_{n-1} : \mathtt{R}_{n-1}(\cdots \Sigma_1 : \mathtt{R}_1(\Sigma_0)\cdots))$$

The *head domain* of Θ is $\mathit{head\text{-}dom}(\Theta) = \mathit{dom}(\Sigma_n)$, the *tail domain* of Θ is $\mathit{tail\text{-}dom}(\Theta) = \bigcup_{i\in 0..n-1} \mathit{dom}(\Sigma_i)$, the *full domain* of Θ is $\mathit{full\text{-}dom}(\Theta) = \mathit{head\text{-}dom}(\Theta) \cup \mathit{tail\text{-}dom}(\Theta)$. We say that the lazy-heap assumption is *well formed* w.r.t. the feature configuration $\overline{\psi}$ to mean that the judgement $\Vdash_{\overline{\psi}} \Theta$ can be derived by the rules in Figure 15. All the lazy-heap assumptions mentioned in the rest of this paper are well-formed w.r.t. a feature configuration that is either explicitly mentioned or understood from the context. According to rule (WF-HEAPENV), a heap assumption Σ is well formed in the configuration $\overline{\psi}$ if every class mentioned in Σ is defined in $\overline{\psi}$. A lazy heap assumption $\Sigma : \mathtt{R}(\Theta)$ is well formed in the configuration $\overline{\psi}$ if so is Σ and if Θ is well formed in the target reconfiguration $\overline{\psi}'$ of $\mathtt{R}$. Additionally, all addresses in $\mathit{dom}(\Sigma)$ that are not in the domain of the topmost heap in Θ must refer to objects created after any other object in Θ, and the correspondence between the class of the objects whose address in Σ and the class of the same objects in Θ is given by $\mathtt{R}$.

As reductions may create and reconfigure objects, (lazy) heap assumptions need to be updated accordingly. To this aim, we define the relation $\ni$ between lazy heap assumptions inductively as follows:

$$\frac{\Sigma \supseteq \Sigma'}{\Sigma \ni \Sigma'} \qquad \frac{\Sigma \supseteq \Sigma' \qquad \Theta \ni \Theta'}{\Sigma : \mathtt{R}(\Theta) \ni \Sigma' : \mathtt{R}(\Theta')}$$

Evaluation contexts, which reflect the congruence rules (see Figure 13), are defined as follows:

$$E ::= [\,] \mid E.\mathtt{f} \mid E.\mathtt{m}(\overline{\mathtt{e}}) \mid \mathtt{v}.\mathtt{m}(\overline{\mathtt{v}}, E, \overline{\mathtt{e}}) \mid E.\mathtt{f} = \mathtt{e} \mid \mathtt{v}.\mathtt{f} = E \mid \mathtt{return}(E)$$

A run-time expression e is *well formed* w.r.t. a stack $\overline{\iota} = \iota_1 \iota_2 \cdots \iota_n$ $(n \geq 0)$, written $\mathbf{wf}(\overline{\iota}, e)$, if e is of the form

$$E_1[\mathtt{return}(E_2[\mathtt{return}(\cdots E_n[\mathtt{return}(e)]\cdots)])]$$

where $E_1, \ldots, E_n, e$ do not contain occurrences of `return`.

Typing rules for runtime expressions are shown in Figure 16; these rules are of the shape $\Theta \vdash_{\overline{\psi}} \overline{\iota}, e : \mathtt{T}$. In Figure 16 we also present the notion of *well-formed lazy heap* and of *well-formed state*. The notion of well-formed lazy heap ensures that the environment Θ maps all the addresses in the lazy heap into the type of the corresponding object and that for every object stored in the lazy heap, the fields of the object contain appropriate values.

The next lemma states that the *olookup* function returns an object of the expected type and a new lazy heap that is well typed if so is the original lazy heap. In fact, the statement of the lemma involves a number of auxiliary functions of Figure 14 as these functions are mutually recursive and therefore it is necessary to prove their correctness collectively. Well-foundedness of these functions (and of the proof of the lemma) is guaranteed by the fact that a lazy heap consists of a *finite* number of reconfigurations.

Lemma 1 *Let* **(*)** $\Theta \Vdash_{\overline{\psi}} \mathscr{L}$ *and suppose* $\vdash \mathtt{R}$ **ok** *for all clauses* $\mathtt{R}$ *occurring in* $\mathscr{L}$*. Then:*

1. *if* $\mathit{olookup}(\iota, \mathscr{L}) = \mathtt{o}, \mathscr{L}'$*, then there exists* $\Theta' \ni \Theta$ *such that* $\Theta' \Vdash_{\overline{\psi}} \mathscr{L}'$ *and* $\mathscr{L}'(\iota) = \mathtt{o}$*;*
2. *if* $\mathit{oreconf}(\iota, \mathtt{R}, \mathscr{L}) = \mathtt{o}, \mathscr{H}, \mathscr{L}'$*, then there exist* Σ *and* $\Theta' \ni \Theta$ *such that* $\{\iota : \mathtt{R}(\mathit{clookup}(\iota, \Theta))\}, \Sigma : \mathtt{R}(\Theta') \Vdash_{\overline{\psi}} \{\iota \mapsto \mathtt{o}\}, \mathscr{H} : \mathtt{R}(\mathscr{L}')$ *where* $\mathtt{R} = \overline{\psi}' \Rightarrow \overline{\psi}\{\cdots\}$*;*
3. *if* $\overline{\psi}; \mathit{clookup}(\iota, \Theta) \vdash \mathtt{A}\,\mathtt{y} = \mathtt{p}$ **ok** *and* $\mathit{preeval}(\mathtt{p}[\iota/\mathtt{this}], \mathscr{L}) = \mathtt{v}, \mathscr{L}'$*, then there exists* $\Theta' \ni \Theta$ *such that* $\Theta' \vdash_{\overline{\psi}} \mathtt{v} : \mathtt{D}$ *and* $\mathtt{D} <:_{\overline{\psi}} \mathtt{A}$*;*
4. *if* $\overline{\psi}; \mathit{clookup}(\iota, \Theta) \vdash \mathtt{p} : \mathtt{D}$ *and* $\mathit{preeval}(\mathtt{p}[\iota/\mathtt{this}], \mathscr{L}) = \mathtt{v}, \mathscr{L}'$*, then there exists* $\Theta' \ni \Theta$ *such that* $\Theta' \Vdash_{\overline{\psi}} \mathscr{L}'$ *and* $\Theta' \vdash_{\overline{\psi}} \mathtt{v} : \mathtt{D}'$ *and* $\mathtt{D}' <:_{\overline{\psi}} \mathtt{D}$*.*

Proof We prove all items simultaneously, and we proceed by induction on the depth of $\mathscr{L}$, where the depth of a lazy heap is the number of reconfigurations that occur in it (an heap $\mathscr{H}$ has depth 0):

1. In the base case it must be $\mathscr{L} = \mathscr{H}$ for some $\mathscr{H}$ and $\mathtt{o} = \mathscr{H}(\iota)$. We conclude immediately by taking $\Theta' = \mathscr{H}$.
   In the inductive case we have $\mathscr{L} = \mathscr{H} : \mathtt{R}(\mathscr{L}_1)$ where $\mathtt{R} = \overline{\psi}' \Rightarrow \overline{\psi}\{\cdots\}$ and we distinguish two subcases. If $\iota \in \mathit{dom}(\mathscr{H})$, then $\mathtt{o} = \mathscr{H}(\iota)$ and we conclude immediately by taking $\Theta' = \Theta$. Suppose $\iota \notin \mathit{dom}(\mathscr{H})$. Then **(O1)** $\mathit{olookup}(\iota, \mathscr{L}_1) = _, \mathscr{L}_1'$ and **(O2)** $\mathit{oreconf}(\iota, \mathtt{R}, \mathscr{L}_1') = \mathtt{o}, \mathscr{H}', \mathscr{L}_1''$ and $\mathscr{L}' = \mathscr{H} \cup \{\iota \mapsto \mathtt{o}\} \cup \mathscr{H}' : \mathtt{R}(\mathscr{L}_1'')$. From **(*)** we deduce $\Theta = \Sigma : \mathtt{R}(\Theta_1)$ for some Σ and Θ_1 such that $\Theta_1 \Vdash_{\overline{\psi}'} \mathscr{L}_1$. From **(O1)** and the induction hypothesis we deduce that there exist $\Theta_1' \ni \Theta_1$ such that $\Theta_1' \Vdash_{\overline{\psi}'} \mathscr{L}_1'$. From **(O2)** and item (2) we deduce that there exist Σ' and $\Theta_1'' \ni \Theta_1'$ such that $\{\iota : \mathtt{R}(\mathit{clookup}(\iota, \Theta))\}, \Sigma' : \mathtt{R}(\Theta_1'') \Vdash_{\overline{\psi}} \{\iota \mapsto \mathtt{o}\}, \mathscr{H} : \mathtt{R}(\mathscr{L}_1'')$. We conclude by taking $\Theta' = \Sigma \cup \{\iota : \mathit{clookup}(\iota, \Theta))\} \cup \Sigma' : \mathtt{R}(\Theta_1'')$.
2. Follows from item (3) and Lemma 3.
3. From (T-PREASSIGN) we deduce $\overline{\psi}; \mathit{clookup}(\iota, \Theta) \vdash \mathtt{p} : \mathtt{A}'$ and $\mathtt{A}' <:_{\overline{\psi}} \mathtt{A}$. By item (4) we deduce that there exists $\Theta' \ni \Theta$ such that $\Theta' \vdash_{\overline{\psi}} \mathtt{v} : \mathtt{D}$ with $\mathtt{D} <:_{\overline{\psi}} \mathtt{A}'$. We conclude by transitivity of $<:_{\overline{\psi}}$.
4. We proceed by induction on the structure of $\mathtt{p}$ and by cases on the rule for *preeval* applied:

Runtime expression typing:

$$\text{(RT-VAR)}\quad \Theta \vdash_{\overline{\psi}} \iota : \mathit{clookup}(\iota,\Theta) \qquad \text{(RT-NEW)}\quad \frac{\mathtt{C} \in \mathit{dom}(\mathtt{CT}_{\overline{\psi}})}{\Theta \vdash_{\overline{\psi}} \mathtt{new\ C}() : \mathtt{C}}$$

$$\text{(RT-NULL)}\quad \frac{\mathtt{T} \in \{\bot\} \cup \{\mathtt{Object}\} \cup \mathit{dom}(\mathtt{CT}_{\overline{\psi}})}{\Theta \vdash_{\overline{\psi}} \mathtt{null} : \mathtt{T}}$$

$$\text{(RT-FIELD)}\quad \frac{\Theta \vdash_{\overline{\psi}} e : \mathtt{C} \qquad \mathtt{A\ f} \in \mathit{fields}_{\overline{\psi}}(\mathtt{C})}{\Theta \vdash_{\overline{\psi}} e.\mathtt{f} : \mathtt{A}}$$

$$\text{(RT-INVK)}\quad \frac{\Theta \vdash_{\overline{\psi}} e : \mathtt{C} \qquad \mathit{meth}_{\overline{\psi}}(\mathtt{m},\mathtt{C}) = \mathtt{B\ m}(\bar{\mathtt{A}}_)_ \qquad \Theta \vdash_{\overline{\psi}} \overline{e} : \overline{\mathtt{T}} \qquad \overline{\mathtt{T}} <:_{\overline{\psi}} \bar{\mathtt{A}}}{\Theta \vdash_{\overline{\psi}} e.\mathtt{m}(\overline{e}) : \mathtt{B}}$$

$$\text{(RT-ASSIGN)}\quad \frac{\Theta \vdash_{\overline{\psi}} e_0.\mathtt{f} : \mathtt{C} \qquad \Theta \vdash_{\overline{\psi}} e_1 : \mathtt{T} \qquad \mathtt{T} <:_{\overline{\psi}} \mathtt{C}}{\Theta \vdash_{\overline{\psi}} e_0.\mathtt{f} = e_1 : \mathtt{C}}$$

$$\text{(RT-RETURN)}\quad \frac{\Theta \vdash_{\overline{\psi}} e : \mathtt{C}}{\Theta \vdash_{\overline{\psi}} \mathtt{return}(e) : \mathtt{C}}$$

Well-formed lazy heap:

$$\text{(WF-HEAP)}\quad \frac{\mathit{dom}(\mathscr{H}) = \mathit{dom}(\Sigma) \qquad (\mathscr{H}(\iota) = \langle \mathtt{C}, \mathtt{f}_1 = \mathtt{v}_1, \ldots, \mathtt{f}_n = \mathtt{v}_n \rangle \quad \textbf{implies} \\ \Sigma(\iota) = \mathtt{C} \qquad \mathit{fields}_{\overline{\psi}}(\mathtt{C}) = \mathtt{C}_1\ \mathtt{f}_1, \ldots, \mathtt{C}_n\ \mathtt{f}_n \qquad \Sigma \vdash_{\overline{\psi}} \mathtt{v}_i : \mathtt{T}_i\ {}^{(i\in 1..n)} \qquad \mathtt{T}_i <:_{\overline{\psi}} \mathtt{C}_i\ {}^{(i\in 1..n)}\,)\ {}^{(\iota \in \mathit{dom}(\mathscr{H}))}}{\Sigma \Vdash_{\overline{\psi}} \mathscr{H}}$$

$$\text{(WF-LAZYHEAP)}\quad \frac{\mathit{dom}(\mathscr{H}) = \mathit{dom}(\Sigma) \\ \mathtt{R} = \overline{\psi}' \Rightarrow \overline{\psi}\ \{_\} \qquad \Theta \Vdash_{\overline{\psi}'} \mathscr{L} \qquad (\mathscr{H}(\iota) = \langle \mathtt{C}, \mathtt{f}_1 = \mathtt{v}_1, \ldots, \mathtt{f}_n = \mathtt{v}_n \rangle \quad \textbf{implies} \\ \Sigma(\iota) = \mathtt{C} \qquad \mathit{fields}_{\overline{\psi}}(\mathtt{C}) = \mathtt{C}_1\ \mathtt{f}_1, \ldots, \mathtt{C}_n\ \mathtt{f}_n \qquad \Sigma : \mathtt{R}(\Theta) \vdash_{\overline{\psi}} \mathtt{v}_i : \mathtt{T}_i\ {}^{(i\in 1..n)} \qquad \mathtt{T}_i <:_{\overline{\psi}} \mathtt{C}_i\ {}^{(i\in 1..n)}\,)\ {}^{(\iota \in \mathit{dom}(\mathscr{H}))}}{\Sigma : \mathtt{R}(\Theta) \Vdash_{\overline{\psi}} \mathscr{H} : \mathtt{R}(\mathscr{L})}$$

Well-formed state:

$$\text{(WF-CONF)}\quad \frac{\overline{\iota} \subseteq \mathit{head\text{-}dom}(\Theta) \qquad \mathbf{wf}(\overline{\iota}, e) \qquad \Theta \Vdash_{\overline{\psi}} \mathscr{L} \qquad \Theta \vdash_{\overline{\psi}} e : \mathtt{T}}{\Theta \vdash_{\overline{\psi}} \mathscr{L}, \overline{\iota}, e : \mathtt{T}}$$

Fig. 16 Typing rules for runtime expressions, lazy heaps and states for IMPERATIVE FEATHERWEIGHT DYNAMIC DELTA JAVA (IFDΔJ).

- ($\mathtt{p} = \mathtt{this}$) Then $\mathtt{v} = \iota$ and we conclude by taking $\Theta' = \Theta$ and $\mathtt{D} = \mathtt{D}' = \mathit{clookup}(\iota, \Theta)$.
- ($\mathtt{p} = \mathtt{p}'.\mathtt{f}$ and $\mathit{preeval}(\mathtt{p}'[\iota/\mathtt{this}], \mathscr{L}) = \mathtt{null}, \mathscr{L}'$). Then $\mathtt{v} = \mathtt{null}$. By induction hypothesis there exists $\Theta' \ni \Theta$ such that $\Theta' \Vdash_{\overline{\psi}} \mathscr{L}'$. We conclude by taking $\mathtt{D}' = \mathtt{D}$.
- ($\mathtt{p} = \mathtt{p}'.\mathtt{f}$ and $\mathit{preeval}(\mathtt{p}'[\iota/\mathtt{this}], \mathscr{L}) = \iota', \mathscr{L}''$ and $\mathit{olookup}(\iota', \mathscr{L}'') = \langle \mathtt{C}', \overline{\mathtt{f}} = \overline{\mathtt{u}} \rangle, \mathscr{L}'$). From rule (T-PREEXPFIELD) we deduce $\overline{\psi}; \mathit{clookup}(\iota, \Theta) \vdash \mathtt{p}' : \mathtt{D}_0$ and $\mathtt{D\,f} \in \mathit{fields}_{\overline{\psi}}(\mathtt{D}_0)$. By induction hypothesis (on $\mathtt{p}'$) there exists $\Theta'' \ni \Theta$ such that $\Theta'' \vdash \iota' : \mathtt{D}'_0$ and $\mathtt{D}'_0 <:_{\overline{\psi}} \mathtt{D}_0$, therefore $\mathtt{D\,f} \in \mathit{fields}_{\overline{\psi}}(\mathtt{D}'_0)$.

By induction hypothesis (on the depth of Θ'') there exists $\Theta' \ni \Theta''$ such that $\Theta' \Vdash_{\overline{\psi}} \mathscr{L}'$. From (*) and (WF-HEAP) we conclude $\Theta' \vdash_{\overline{\psi}} \mathtt{v} : \mathtt{D}'$ for some $\mathtt{D}' <:_{\overline{\psi}} \mathtt{D}$.

The next two lemmas prove fundamental properties about the post-reconfiguration clauses. As the code in these clauses can only create objects in the current feature configuration, the lazy heap is unaffected by their execution except for the topmost level.

Lemma 2 *Let* $\mathtt{R} = \overline{\psi} \Rightarrow \overline{\psi}'\{\cdots\}$ *and:*

1. $\Sigma_1 \vdash \mathtt{v}_i : \mathtt{C}_i$ *and* $\mathtt{C}_i <:_{\overline{\psi}} \mathtt{A}_i$ *for every* $i \in 1..|\overline{\mathtt{v}}|$*;*
2. $\Sigma_2 \vdash \mathtt{u}_i : \mathtt{D}_i$ *and* $\mathtt{D}_i <:_{\overline{\psi}'} \mathtt{B}_i$ *for every* $i \in 1..|\overline{\mathtt{u}}|$*;*
3. $\mathtt{R}; \overline{\mathtt{y} : \mathtt{A}}; \overline{\mathtt{z} : \mathtt{B}} \vdash \mathtt{B}\,\mathtt{z} = \mathtt{q}$ **ok***;*
4. *posteval*$(\overline{\psi}', \mathtt{q}[\overline{\mathtt{v}}/\overline{\mathtt{y}}][\overline{\mathtt{u}}/\overline{\mathtt{z}}]) = \mathtt{u}, \mathscr{H}$.

Then there exists Σ *such that:*

- $\mathtt{R}(\Sigma_1), \Sigma_2, \Sigma \vdash_{\overline{\psi}'} \mathscr{H}$*;*
- $\mathtt{R}(\Sigma_1), \Sigma_2, \Sigma \vdash \mathtt{u} : \mathtt{D}$ *and* $\mathtt{D} <:_{\overline{\psi}'} \mathtt{B}$.

Proof By cases on the shape of $\mathtt{q}$.

- ($\mathtt{q} = \mathtt{null}$) We conclude by taking $\Sigma = \bullet$ and $\mathtt{D} = \mathtt{B}$.
- ($\mathtt{q} = \mathtt{y}_i$) By hypothesis we have $\Sigma_1 \vdash \mathtt{v}_i : \mathtt{C}_i$ and $\mathtt{C}_i <:_{\overline{\psi}} \mathtt{A}_i$. From (T-POSTASSIGNVAR) we deduce $\mathtt{R}(\Sigma_1) \vdash \mathtt{v}_i : \mathtt{R}(\mathtt{C}_i)$ and $\mathtt{R}(\mathtt{C}_i) <:_{\overline{\psi}'} \mathtt{B}$. We conclude by taking $\Sigma = \bullet$ and $\mathtt{D} = \mathtt{R}(\mathtt{C}_i)$.
- ($\mathtt{q} = \mathtt{new}\ \mathtt{C}(\mathtt{z}_1, \ldots, \mathtt{z}_n)$) Then $\mathtt{u} = \iota$ and $\mathscr{H} = \{\iota \mapsto \langle \mathtt{C}, \mathtt{f}_1 = \mathtt{z}_1[\overline{\mathtt{u}}/\overline{\mathtt{z}}], \ldots, \mathtt{f}_n = \mathtt{z}_n[\overline{\mathtt{u}}/\overline{\mathtt{z}}]\rangle\}$. Let $\Sigma = \{\iota : \mathtt{C}\}$ and $\mathtt{D} = \mathtt{C}$ and $\mathtt{D}_1\,\mathtt{f}_1, \ldots, \mathtt{D}_n\,\mathtt{f}_n = \mathit{fields}_{\overline{\psi}'}(\mathtt{C})$. From rule (T-POSTASSIGNNEW) we have $\mathtt{D} = \mathtt{C} <:_{\overline{\psi}'} \mathtt{B}$. Also, from the same rule we deduce that for every $i \in 1..n$ we have $\mathtt{z}_i : \mathtt{B}_i \in \overline{\mathtt{z} : \mathtt{B}}$ and $\mathtt{B}_i <:_{\overline{\psi}'} \mathtt{D}_i$. Now $\mathtt{z}_i[\overline{\mathtt{u}}/\overline{\mathtt{z}}] = \mathtt{u}_j$ for some $j \in 1..|\overline{\mathtt{u}}|$. By hypothesis we have $\Sigma_2 \vdash \mathtt{u}_j : \mathtt{D}_j$ and $\mathtt{D}_j <:_{\overline{\psi}'} \mathtt{B}_j$. We conclude $\mathtt{R}(\Sigma_1), \Sigma_2, \Sigma \vdash_{\overline{\psi}'} \mathscr{H}$ by transitivity of $<:_{\overline{\psi}}$.

Lemma 3 *Let* $\mathtt{R} = \overline{\psi} \Rightarrow \overline{\psi}'\{\cdots\}$ *and:*

1. $\Sigma_1 \vdash \mathtt{v}_i : \mathtt{C}_i$ *and* $\mathtt{C}_i <:_{\overline{\psi}} \mathtt{A}_i$ *for every* $i \in 1..|\overline{\mathtt{v}}|$*;*
2. $\Sigma_2 \vdash \mathtt{u}_i : \mathtt{D}_i$ *and* $\mathtt{D}_i <:_{\overline{\psi}'} \mathtt{B}_i$ *for every* $i \in 1..|\overline{\mathtt{u}}|$*;*
3. $\mathtt{R}; \overline{\mathtt{y} : \mathtt{A}}; \overline{\mathtt{z} : \mathtt{B}} \vdash \overline{\mathtt{B}'\,\mathtt{z}' = \mathtt{q}}$ **ok***;*
4. *postassign*$(\overline{\psi}', \overline{\mathtt{B}'\,\mathtt{z}' = \mathtt{q}}[\overline{\mathtt{v}}/\overline{\mathtt{y}}][\overline{\mathtt{u}}/\overline{\mathtt{z}}]) = \overline{\mathtt{u}}', \mathscr{H}'$*;*

Then there exists Σ *such that:*

- $\mathtt{R}(\Sigma_1), \Sigma_2, \Sigma \vdash_{\overline{\psi}'} \mathscr{H}$*;*
- $\mathtt{R}(\Sigma_1), \Sigma_2, \Sigma \vdash \mathtt{u}'_i : \mathtt{D}'_i$ *and* $\mathtt{D}'_i <:_{\overline{\psi}'} \mathtt{B}'_i$ *for every* $i \in 1..|\overline{\mathtt{u}}'|$.

Proof By induction on $|\overline{\mathtt{u}}'|$. If $|\overline{\mathtt{u}}'| = 0$, then $\mathscr{H} = \emptyset$ and we conclude immediately by taking $\Sigma = \bullet$. Suppose $|\overline{\mathtt{u}}'| > 0$. Then $\overline{\mathtt{B}'\,\mathtt{z}' = \mathtt{q}} = \mathtt{B}'\,\mathtt{z}' = \mathtt{q}\overline{\mathtt{B}''\,\mathtt{z}'' = \mathtt{q}'}$ and $\overline{\mathtt{u}}' = \mathtt{u}'\overline{\mathtt{u}}''$ where

- *posteval*$(\overline{\psi}', \mathtt{q}'[\overline{\mathtt{v}}/\overline{\mathtt{y}}][\overline{\mathtt{u}}/\overline{\mathtt{z}}]) = \mathtt{u}', \mathscr{H}$
- *postassign*$(\overline{\psi}', \overline{\mathtt{B}''\,\mathtt{z}'' = \mathtt{q}'}[\overline{\mathtt{v}}/\overline{\mathtt{y}}][\overline{\mathtt{u}}\mathtt{u}'/\overline{\mathtt{z}}\mathtt{z}']) = \overline{\mathtt{u}}'', \mathscr{H}'$

From Lemma 2 we deduce that there exists Σ' such that $\mathtt{R}(\Sigma_1), \Sigma_2, \Sigma' \vdash_{\overline{\psi}'} \mathscr{H}$ and $\mathtt{R}(\Sigma_1), \Sigma_2, \Sigma' \vdash \mathtt{u}' : \mathtt{D}'$ and $\mathtt{D}' <:_{\overline{\psi}'} \mathtt{B}'$.

By induction hypothesis we deduce that there exists Σ'' such that $\mathtt{R}(\Sigma_1), \Sigma_2, \Sigma'' \vdash_{\overline{\psi}'} \mathscr{H}'$ and for every $i \in 1..|\overline{\mathtt{u}}''|$ we have $\mathtt{R}(\Sigma_1), \Sigma_2, \Sigma'' \vdash \mathtt{u}''_i : \mathtt{D}'_i$ and $\mathtt{D}'_i <:_{\overline{\psi}'} \mathtt{B}'_i$.

We conclude by taking $\Sigma = \Sigma', \Sigma''$.

We are now ready to formally state the subject reduction theorem, whose proof relies upon the auxiliary lemmas just presented.

Theorem 2 **(Subject reduction)** *Let* $\Theta \vdash_{\overline{\psi}} \mathscr{L}, \overline{\iota}, e : \mathtt{T}$ *and* $\mathit{meth}_{\overline{\psi}}(\mathit{main}, \mathit{Main}) = \mathtt{C}_0\ \mathit{main}()\{_\}$*. Then:*

1. *if* $\overline{\psi}, \mathscr{L}, \overline{\iota}, e \longrightarrow \overline{\psi}, \mathscr{L}', \overline{\iota}', e'$*, then there exist* $\Theta' \ni \Theta$ *and* $\mathtt{T}' <:_{\overline{\psi}} \mathtt{T}$ *such that* $\Theta' \vdash_{\overline{\psi}} \mathscr{L}', \overline{\iota}', e' : \mathtt{T}'$.
2. *if* $\overline{\psi}, \mathscr{L}, \overline{\iota}, e \Longrightarrow \overline{\psi}', \emptyset : \mathtt{R}(\mathscr{L}), \overline{\iota}, e$*, then there exists* $\mathtt{T}' <:_{\overline{\psi}'} \mathtt{C}_0$ *such that* $\bullet : \mathtt{R}(\Theta) \vdash_{\overline{\psi}'} (\emptyset : \mathtt{R}(\mathscr{L})), \overline{\iota}, e : \mathtt{T}'$.

Proof The proof of item (1) is almost standard, the only notable exception being the fact that heap is accessed through the auxiliary function $olookup(_,_)$ which may trigger object reconfiguration. Lemma 1 guarantees that the object returned by $olookup(_,_)$ has the right type and that the new heap is still well formed in the current configuration. Item (2) is obvious, since the new heap environment is updated according to the structure of the new heap and the **Enabled** predicate guarantees that the runtime expression e is still well typed in the new configuration.

The progress theorem and its proof are standard.

Theorem 3 (Progress) *Let $\Theta \vdash_{\overline{\psi}} \mathscr{L}, \overline{\iota}, e : \mathtt{T}$ and $\overline{\psi}, \mathscr{L}, \overline{\iota}, e \not\rightarrow$. Then:*

1. *either e is a value, or*
2. *for some evaluation context E we can express e as*
   (a) $E[\mathtt{null.f}]$ *for some* $\mathtt{f}$*, or*
   (b) $E[\mathtt{null.f} = \mathtt{v}]$ *for some* $\mathtt{f}$ *and* $\mathtt{v}$*, or*
   (c) $E[\mathtt{null.m}(\overline{\mathtt{v}})]$ *for some* $\mathtt{m}$ *and* $\overline{\mathtt{v}}$*.*

We conclude with type soundness, which is a straightforward corollary of subject reduction and progress.

PROOF OF THEOREM 1 (TYPE SOUNDNESS). Immediate by Theorems 2 and 3.

References

Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 1015–1022, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-768-4. doi: 10.1145/1639950.1640073. URL `http://doi.acm.org/10.1145/1639950.1640073`.

Vander Alves, Pedro Matos Jr, Leonardo Cole, Alexandre Vasconcelos, Paulo Borba, and Geber Ramalho. Extracting and Evolving Code in Product Lines with Aspect-Oriented Programming. In *Transactions on aspect-oriented software development IV*, pages 117–142. Springer, 2007.

Davide Ancona, Christopher Anderson, Ferruccio Damiani, Sophia Drossopoulou, Paola Giannini, and Elena Zucca. A provenly correct translation of fickle into java. *ACM Trans. Program. Lang. Syst.*, 29(2), April 2007. ISSN 0164-0925. doi: 10.1145/1216374.1216381. URL `http://doi.acm.org/10.1145/1216374.1216381`.

Rodrigo Andrade, Henrique Rebêlo, Márcio Ribeiro, and Paulo Borba. Flexible feature binding with aspectj-based idioms. *J. UCS*, 20(5):692–719, 2014.

Sven Apel, Thomas Leich, Marko Rosenmller, and Gunter Saake. Featurec++: On the symbiosis of feature-oriented and aspect-oriented programming. In Robert Glück and Michael Lowry, editors, *Generative Programming and Component Engineering*, volume 3676 of *Lecture Notes in Computer Science*, pages 125–140. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-29138-1. doi: 10.1007/11561347_10. URL `http://dx.doi.org/10.1007/11561347_10`.

Sven Apel, Christian Kästner, Armin Gröβlinger, and Christian Lengauer. Type safety for feature-oriented product lines. *Automated Software Engg.*, 17(3):251–300, September 2010. ISSN 0928-8910. doi: 10.1007/s10515-010-0066-8. URL `http://dx.doi.org/10.1007/s10515-010-0066-8`.

Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of caesarj. In Awais Rashid and Mehmet Aksit, editors, *Transactions on Aspect-Oriented Software Development I*, volume 3880 of *Lecture Notes in Computer Science*, pages 135–173. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-32972-5. doi: 10.1007/11687061_5. URL `http://dx.doi.org/10.1007/11687061_5`.

Don Batory. Feature models, grammars, and propositional formulas. In Henk Obbink and Klaus Pohl, editors, *Software Product Lines*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-28936-4. doi: 10.1007/11554844_3. URL `http://dx.doi.org/10.1007/11554844_3`.

Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004. ISSN 0098-5589. doi: http://doi.ieeecomputersociety.org/10.1109/TSE.2004.23.

Lorenzo Bettini and Viviana Bono. Type safe dynamic object delegation in class-based languages. In *Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java*, PPPJ '08, pages 171–180, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-223-8. doi: 10.1145/1411732.1411756. URL `http://doi.acm.org/10.1145/1411732.1411756`.

Lorenzo Bettini, Viviana Bono, and Betti Venneri. Delegation by object composition. *Science of Computer Programming*, 76(11):992 – 1014, 2011. ISSN 0167-6423. doi: http://dx.doi.org/10.1016/j.scico.2010.04.006. URL http://www.sciencedirect.com/science/article/pii/S0167642310000754.

Lorenzo Bettini, Sara Capecchi, and Ferruccio Damiani. On flexible dynamic trait replacement for java-like languages. *Science of Computer Programming*, 78(7):907 – 932, 2013a. ISSN 0167-6423. doi: http://dx.doi.org/10.1016/j.scico.2012.11.003. URL http://www.sciencedirect.com/science/article/pii/S0167642312002092.

Lorenzo Bettini, Ferruccio Damiani, and Ina Schaefer. Compositional type checking of delta-oriented software product lines. *Acta Informatica*, 50:77–122, 2013b. ISSN 0001-5903. doi: 10.1007/s00236-012-0173-z. URL http://dx.doi.org/10.1007/s00236-012-0173-z.

Lorenzo Bettini, Ferruccio Damiani, and Ina Schaefer. Implementing type-safe software product lines using parametric traits. *Science of Computer Programming*, 2013c. ISSN 0167-6423. doi: 10.1016/j.scico.2013.07.016. URL http://www.sciencedirect.com/science/article/pii/S0167642313001901.

Richard Bubel, Ferruccio Damiani, Reiner Hähnle, Einar Broch Johnsen, Olaf Owe, Ina Schaefer, and Ingrid Chieh Yu. *Proof Repositories for Compositional Verification of Evolving Software Systems*, pages 130–156. Springer International Publishing, Cham, 2016. ISBN 978-3-319-46508-1. doi: 10.1007/978-3-319-46508-1_8. URL http://dx.doi.org/10.1007/978-3-319-46508-1_8.

Rafael Capilla, Jan Bosch, Pablo Trinidad, Antonio Ruiz-Corts, and Mike Hinchey. An overview of dynamic software product line architectures and techniques: Observations from research and industry. *Journal of Systems and Software*, 91(0):3 – 23, 2014. ISSN 0164-1212. doi: 10.1016/j.jss.2013.12.038. URL http://www.sciencedirect.com/science/article/pii/S0164121214000119.

Susanne Cech Previtali and Thomas R. Gross. Aspect-based dynamic software updating: a model and its empirical evaluation. In *AOSD*, pages 105–116, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0605-8. doi: 10.1145/1960275.1960289. URL http://doi.acm.org/10.1145/1960275.1960289.

Venkat Chakravarthy, John Regehr, and Eric Eide. Edicts: Implementing features with flexible binding times. In *Proceedings of the 7th International Conference on Aspect-oriented Software Development*, AOSD '08, pages 108–119, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-044-9. doi: 10.1145/1353482.1353496. URL http://doi.acm.org/10.1145/1353482.1353496.

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991. ISSN 0164-0925. doi: 10.1145/115372.115320. URL http://doi.acm.org/10.1145/115372.115320.

Ferruccio Damiani and Michael Lienhardt. *On Type Checking Delta-Oriented Product Lines*, pages 47–62. Springer International Publishing, Cham, 2016. ISBN 978-3-319-33693-0. doi: 10.1007/978-3-319-33693-0_4. URL http://dx.doi.org/10.1007/978-3-319-33693-0_4.

Ferruccio Damiani and Ina Schaefer. Dynamic delta-oriented programming. In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, SPLC '11, pages 34:1–34:8, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0789-5. doi: 10.1145/2019136.2019175. URL http://doi.acm.org/10.1145/2019136.2019175.

Ferruccio Damiani and Ina Schaefer. *Family-Based Analysis of Type Safety for Delta-Oriented Software Product Lines*, pages 193–207. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-34026-0. doi: 10.1007/978-3-642-34026-0_15. URL http://dx.doi.org/10.1007/978-3-642-34026-0_15.

Ferruccio Damiani, Sophia Drossopoulou, and Paola Giannini. Refined effects for unanticipated object reclassification: Fickle$_3$. In Carlo Blundo and Cosimo Laneve, editors, *Theoretical Computer Science*, volume 2841 of *Lecture Notes in Computer Science*, pages 97–110. Springer Berlin Heidelberg, 2003. ISBN 978-3-540-20216-5. doi: 10.1007/978-3-540-45208-9_9. URL http://dx.doi.org/10.1007/978-3-540-45208-9_9.

Ferruccio Damiani, Olaf Owe, Johan Dovland, Ina Schaefer, Einar B. Johnsen, and Ingrid C. Yu. A transformational proof system for delta-oriented programming. In *Proceedings of the 16th International Software Product Line Conference - Volume 2*, SPLC '12, pages 53–60, New York, NY, USA, 2012a. ACM. ISBN 978-1-4503-1095-6. doi: 10.1145/2364412.2364422. URL http://doi.acm.org/10.1145/2364412.2364422.

Ferruccio Damiani, Luca Padovani, and Ina Schaefer. A formal foundation for dynamic delta-oriented software product lines. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*, GPCE '12, pages 1–10, New York, NY, USA, 2012b. ACM. ISBN 978-1-4503-1129-8. doi: 10.1145/2371401.2371403. URL http://doi.acm.org/10.1145/2371401.2371403.

Ferruccio Damiani, Christoph Gladisch, and Shmuel Tyszberowicz. Refinement-based testing of delta-oriented product lines. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '13, pages 135–

140, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2111-2. doi: 10.1145/2500828.2500841. URL http://doi.acm.org/10.1145/2500828.2500841.

Tom Dinkelaker, Ralf Mitschke, Karin Fetzer, and Mira Mezini. A dynamic software product line approach using aspect models at runtime. In *Fifth Domain-Specific Aspect Languages Workshop*, volume 39, page 40, 2010.

Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. More dynamic object reclassification: Fickle∥. *ACM Trans. Program. Lang. Syst.*, 24(2):153–191, March 2002. ISSN 0164-0925. doi: 10.1145/514952.514955. URL http://doi.acm.org/10.1145/514952.514955.

Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.*, 28(2):331–388, March 2006. ISSN 0164-0925. doi: 10.1145/1119479.1119483. URL http://doi.acm.org/10.1145/1119479.1119483.

Erik Ernst. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Århus, Denmark, 1999. Url: http://www.daimi.au.dk/~eernst/gbeta/.

Iris Groher and Markus Voelter. Aspect-Oriented Model-Driven Software Product Line Engineering. In *Transactions on aspect-oriented software development VI*, pages 111–152. Springer, 2009.

Tianxiao Gu, Chun Cao, Chang Xu, Xiaoxing Ma, Linghao Zhang, and Jian Lü. Low-disruptive dynamic updating of java applications. *Information and Software Technology*, 56(9):1086 – 1098, 2014. ISSN 0950-5849. doi: http://dx.doi.org/10.1016/j.infsof.2014.04.003. URL http://www.sciencedirect.com/science/article/pii/S0950584914000846.

Sebastian Günther and Sagar Sunkle. Dynamically adaptable software product lines using ruby metaprogramming. In *Proceedings of the 2Nd International Workshop on Feature-Oriented Software Development*, FOSD '10, pages 80–87, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0208-1. doi: 10.1145/1868688.1868700. URL http://doi.acm.org/10.1145/1868688.1868700.

Reiner Hähnle and Ina Schaefer. A liskov principle for delta-oriented programming. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, volume 7609 of *Lecture Notes in Computer Science*, pages 32–46. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-34025-3. doi: 10.1007/978-3-642-34026-0_4. URL http://dx.doi.org/10.1007/978-3-642-34026-0_4.

Svein Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. Dynamic software product lines. *Computer*, 41(4):93–95, 2008. ISSN 0018-9162. doi: http://doi.ieeecomputersociety.org/10.1109/MC.2008.123.

Stephan Herrmann. A precise model for contextual roles: The programming language ObjectTeams/Java. *Applied Ontology*, 2(2):181–207, 2007. URL http://iospress.metapress.com/content/M186325145U8166N.

Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, March 2008. ISSN 1660-1769. doi: 10.5381/jot.2008.7.3.a4. URL http://www.jot.fm/contents/issue_2008_03/article4.html.

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: A minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001. ISSN 0164-0925. doi: 10.1145/503502.503505. URL http://doi.acm.org/10.1145/503502.503505.

Einar B. Johnsen, Marcel Kyas, and Ingrid C. Yu. Dynamic classes: Modular asynchronous evolution of distributed concurrent objects. In Ana Cavalcanti and Dennis R. Dams, editors, *FM 2009: Formal Methods*, volume 5850 of *Lecture Notes in Computer Science*, pages 596–611. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-05088-6. doi: 10.1007/978-3-642-05089-3_38. URL http://dx.doi.org/10.1007/978-3-642-05089-3_38.

Christian Kästner, Sven Apel, and Don Batory. A Case Study Implementing Features Using AspectJ. In *Software Product Line Conference*, pages 223–232, Los Alamitos, CA, USA, 2007. IEEE Computer Society. doi: http://doi.ieeecomputersociety.org/10.1109/SPLINE.2007.12.

Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 311–320, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-079-1. doi: 10.1145/1368088.1368131. URL http://doi.acm.org/10.1145/1368088.1368131.

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. *Aspect-Oriented Programming*. Springer, 1997.

Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In Jørgen L. Knudsen, editor, *ECOOP 2001 Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–354. Springer Berlin Heidelberg, 2001. ISBN 978-3-540-42206-8. doi: 10.1007/3-540-45337-7_18. URL http://dx.doi.org/10.1007/

3-540-45337-7_18.

Jonathan Koscielny, Sönke Holthusen, Ina Schaefer, Sandro Schulze, Lorenzo Bettini, and Ferruccio Damiani. Deltaj 1.5: delta-oriented programming for java 1.5. In *PPPJ 2014*, pages 63–74, 2014.

Charles Krueger. Eliminating the adoption barrier. *IEEE Softw.*, 19(4):29–31, July 2002. ISSN 0740-7459. doi: 10.1109/MS.2002.1020284. URL http://dx.doi.org/10.1109/MS.2002.1020284.

Malte Lochau, Ina Schaefer, Jochen Kamischke, and Sascha Lity. Incremental model-based testing of delta-oriented software product lines. In Achim D. Brucker and Jacques Julliand, editors, *Tests and Proofs*, volume 7305 of *Lecture Notes in Computer Science*, pages 67–82. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-30472-9. doi: 10.1007/978-3-642-30473-6_7. URL http://dx.doi.org/10.1007/978-3-642-30473-6_7.

Roberto E. Lopez-Herrejon, Don Batory, and William Cook. Evaluating support for features in advanced modularization technologies. In Andrew P. Black, editor, *ECOOP 2005 - Object-Oriented Programming*, volume 3586 of *Lecture Notes in Computer Science*, pages 169–194. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-27992-1. doi: 10.1007/11531142_8. URL http://dx.doi.org/10.1007/11531142_8.

Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering - Foundations, Principles and Techniques*. Springer Berlin/Heidelberg, 2005.

Mario Pukall, Christian Kästner, Walter Cazzola, Sebastian Götz, Alexander Grebhahn, Reimar Schöter, and Gunter Saake. JavAdaptor — Flexible Runtime Updates of Java Applications. *Software—Practice and Experience*, 43(2):153–185, February 2013. doi: 10.1002/spe.2107.

Márcio Ribeiro, Rodrigo Cardoso, Paulo Borba, Rodrigo Bonifácio, and Henrique Rebêlo. Does aspectj provide modularity when implementing features with flexible binding times? *Third Latin American Workshop on Aspect-Oriented Software Development (LA-WASP 2009), Fortaleza, Ceara, Brazil*, pages 1–6, 2009.

Marko Rosenmüller, Norbert Siegmund, Sven Apel, and Gunter Saake. Flexible feature binding in software product lines. *Automated Software Engineering*, 18(2):163–197, 2011a. ISSN 0928-8910. doi: 10.1007/s10515-011-0080-5. URL http://dx.doi.org/10.1007/s10515-011-0080-5.

Marko Rosenmüller, Norbert Siegmund, Mario Pukall, and Sven Apel. Tailoring dynamic software product lines. *SIGPLAN Not.*, 47(3):3–12, October 2011b. ISSN 0362-1340. doi: 10.1145/2189751.2047866. URL http://doi.acm.org/10.1145/2189751.2047866.

Darpan Saini, Joshua Sunshine, and Jonathan Aldrich. A theory of typestate-oriented programming. In *FTfJP*, pages 9:1–9:7. ACM, 2010. ISBN 978-1-4503-0540-2. doi: http://doi.acm.org/10.1145/1924520.1924529. URL http://doi.acm.org/10.1145/1924520.1924529.

Ina Schaefer and Ferruccio Damiani. Pure delta-oriented programming. In *Proceedings of the 2Nd International Workshop on Feature-Oriented Software Development*, FOSD '10, pages 49–56, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0208-1. doi: 10.1145/1868688.1868696. URL http://doi.acm.org/10.1145/1868688.1868696.

Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In Jan Bosch and Jaejoon Lee, editors, *Software Product Lines: Going Beyond*, volume 6287 of *Lecture Notes in Computer Science*, pages 77–91. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-15578-9. doi: 10.1007/978-3-642-15579-6_6. URL http://dx.doi.org/10.1007/978-3-642-15579-6_6.

Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. Software diversity: state of the art and perspectives. *STTT*, 14(5):477–495, 2012. doi: 10.1007/s10009-012-0253-y. URL http://dx.doi.org/10.1007/s10009-012-0253-y.

Christoph Seidl, Ina Schaefer, and Uwe Aßmann. Integrated Management of Variability in Space and Time in Software Families. In *Proceedings of the 18th International Software Product Line Conference (SPLC)*, SPLC'14, 2014.

Yannis Smaragdakis and Don Batory. Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Methodol.*, 11(2):215–255, April 2002. ISSN 1049-331X. doi: 10.1145/505145.505148. URL http://doi.acm.org/10.1145/505145.505148.

Charles Smith and Sophia Drossopoulou. Chai: Traits for java-like languages. In AndrewP. Black, editor, *ECOOP 2005 - Object-Oriented Programming*, volume 3586 of *Lecture Notes in Computer Science*, pages 453–478. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-27992-1. doi: 10.1007/11531142_20. URL http://dx.doi.org/10.1007/11531142_20.

Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 107–119, New York, NY, USA, 1999. ACM. ISBN 1-58113-074-0. doi: 10.1145/302405.302457. URL http://doi.acm.org/10.1145/302405.302457.

Thomas Würthinger, Christian Wimmer, and Lukas Stadler. Unrestricted and safe dynamic code evolution for java. *Science of Computer Programming*, 78(5):481 – 498, 2013. ISSN 0167-6423. doi: 10.1016/j.scico.2011.06.005. URL http://www.sciencedirect.com/science/article/pii/S0167642311001456.