Row Types for Delta-Oriented Programming*

Michael Lienhardt
University of Bologna, Italy
lienhard@cs.unibo.it

Dave Clarke
Katholieke Universiteit Leuven, Belgium
Dave.Clarke@cs.kuleuven.be

ABSTRACT

Delta-oriented programming (DOP) provides a technique for implementing Software Product Lines based on modifications (add, remove, modify) to a core program. Unfortunately, such modifications can introduce errors into a program, especially when type signatures of classes are modified in a non-monotonic fashion. To deal with this problem we present a type system for delta-oriented programs based on row polymorphism. This exercise elucidates the close correspondence between delta-oriented programs and row polymorphism.

General Terms

Theory

Keywords

Software product line engineering, delta-oriented programming, structural typing

1. INTRODUCTION

Software product line engineering (SPLE) [6] is a powerful paradigm for incrementally building families of programs based on a common *core* collection of artefacts. The basic breakdown of an SPL is quite intuitive: one specifies the core architecture, the *features* that can be added to the core program, a *feature model* of the SPL, which constraints on how the features are applied, such as whether a feature is mandatory or whether features are exclusive, etc. [16], and, for each feature, a *feature module* describing the modifications to the core required to implement the feature. This is called *family engineering*. Finally, one constructs a product—called *application engineering*—by selecting the corresponding (valid) set of features, and generating the desired code based on the core architecture and selected feature modules.

SPLs have been intensively studied, and many different approaches to developing product lines exist. One recent development approach is *delta-oriented programming* [21, 22]. This approach addresses several limitations that constrain the designs developers can write: it enables features to be implemented using more than one feature module, thereby improving the modularity of designs—a `debug` feature could be structured into different modules; it enables feature modules to be applied for different combinations of features, increasing reuse and flexibility—features `colour` and `print` could have associated feature modules, but when used in combination, an additional feature module `coloured_print` would also be incorporated; and it enables non-monotonic modifications of the core architecture, including the removal of fields, methods and even classes—reflection over objects (for marshalling or automatically generating GUI components) will work with the right collection of fields and methods.

Delta-oriented programming addresses the implementation of SPLs using the following ingredients: *deltas*: an extension of feature module that are not bound to a particular feature, which can add, remove, wrap or replace classes, methods and fields; and a mapping from deltas to feature combinations via *application conditions*, which are propositional formula over feature names. This model for SPL is flexible and enables the modular construction of SPLs. It is, however, quite recently developed, and consequently the tool support for DOP is not as mature as for other SPL approaches. In particular, the issue of typing DOP has not fully been resolved. The approach of Schaefer et al. [20] generates a collection of constraints for a delta-oriented product line, but these can only be checked per product. Most importantly, from our perspective, the type system does not reflect the structure of the deltas as types.

Our line of work aims to define an intuitive and modular type system for DOP. Our approach is based on the observation that the operations underlying DOP are similar to operations on records [18] and that these can be typed using a row polymorphic type system. Our type system ensures that the application of deltas to a core program does not add a class or method twice or remove them when they are not present. It also gives a natural type for the composition of two deltas. In the future we will extend our approach to checking that all products in an SPL are type safe.

*This research is partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (`http://www.hats-project.eu`)

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
VaMoS '12, January 25-27, 2012 Leipzig, Germany
Copyright 2012 ACM 978-1-4503-1058-1 ...$10.00.

The paper is structured as follow. Section 2 describes delta-oriented programming, giving examples of deltas and their syntax. Section 3 introduces our approach to typing deltas. Section 4 briefly discusses the limitations of our approach. Section 5 compares our approach with related work and Section 6 concludes the paper.

2. DELTA-ORIENTED PROGRAMMING

In this section, we illustrate DOP with a simple example written in DeltaJava [21], an extension of Java for delta-oriented Programming. Let suppose that we are developing an SPL to control a set of coffee machines: the core product corresponds to the basic coffee machine model, while deltas encode possible variations on this initial model. The core product, named `C`, is presented Figure 1 and consists of a simple coffee machine with the ability to make coffee or tomato juice. The core has two classes `Coffee` and

```
core C {
  class Coffee {
    void fill(int q) { ... }
    void make() { ... }
  }
  class TomatoJuice {
    void fill(int q) { ... }
    void make() { ... }
  }
}
```

Figure 1: Core Program

`TomatoJuice` that respectively handle putting coffee (resp. tomato powder) in the machine with method `fill` and making coffee (resp. tomato juice) with method `make`.

Starting from this core product, we would like to develop a controller for another kind of machine that does not make tomato juice, but does offer sugar and milk and can also mix hot chocolate. To achieve this, we define the three deltas presented Figure 2. The first, `D1`, deals with sugar and milk. It adds one class to the product, `Settings`, with two methods giving the possibility to set the desired quantity of milk and sugar. The second delta, `D2`, removes class `TomatoJuice`, for coffee machines that cannot make juice. Finally, the last delta, `D3`, deals with the hot chocolate feature. This delta first adds a new class `Chocolate`, which has method `fill` for when chocolate powder is put in the machine, and `make` to prepare the hot chocolate. The delta also modifies the class `Settings`, adding two new methods that offer the possibility of either dark or white chocolate.

Applying a delta to a core program is quite intuitive, as it does exactly what the programmer described: classes and interfaces that are introduced by the command **adds** in the delta are added to the program; classes and interfaces that are mentioned by the command **removes** in the delta are removed from the program; and classes and interfaces to which the programmer wants to adds or remove fields and methods with the command **modifies** are modified accordingly. For instance, applying delta `D1` to the core program will result in the program shown in Figure 3a and applying `D1`, `D2` and `D3` in sequence will result in the code shown in Figure 3b.

Manipulations done by deltas are quite simple and powerful. But they can potentially lead to three kinds of errors during delta application: a delta could add an element (class, interface, field or method) that is already present in the input program; a delta could remove an element that does not exist in the input program; and a delta could modify an element that does not exist in the input program. For instance, if we apply `D2`, `D3` and then `D1` to our core program, an error will be raised because `D3` would modify class `Settings`, which is not present in the program—it is added later by `D1`. One can easily see this error as the number and the size of the deltas and core program are quite small. But real SPLs are much larger and cannot be checked by hand.

```
delta D1 {
  adds class Settings {
    void setSugar(int q) { ... }
    void setMilk(int q) { ... }
  }
}

delta D2 {
  removes class TomatoJuice
}

delta D3 {
  adds class Chocolate {
    void fill(int q) { ... }
    void make() { ... }
  }
  modifies class Settings {
     adds void setChocolateDark() { ... }
     adds void setChocolateWhite() { ... }
  }
}
```

Figure 2: Deltas

This paper presents a type system that checks applications of deltas to a program and ensures that none of the previously mentioned errors occurs in a well-typed application.

3. TYPING DELTAS

Our approach focuses on typing deltas, their application on a program, and the composition of deltas. As deltas modify the *structure* of the program, that is, the class and interface bodies, our type system will focus on these elements, and will abstract away other typing considerations, such as inheritance and dependencies between classes. In particular, our approach is not concerned with typing method bodies. Even though these elements are crucial to ensure the type correctness of a program, these are omitted from our approach to highlight the correspondence between deltas and row polymorphism.

3.1 Row Polymorphism and Deltas

A record is a finite set of associations, called fields, between labels and values, which can also be records. Many languages, such as O'Caml, use records as primitive values and provide a wide range of operations on records. Type systems for records [18] guarantee that record operations will never fail, for example, preventing an attempt to remove a field that does not exist.

The parallel between deltas and record operations is quite

```
class Coffee {
  void fill(int q) { ... }
  void make() { ... }
}
class TomatoJuice {
  void fill(int q) { ... }
  void make() { ... }
}
class Settings {
  void setSugar(int q) { ... }
  void setMilk(int q) { ... }
}
```

(a) After the application of `D1`

```
class Coffee {
  void fill(int q) { ... }
  void make() { ... }
}
class Settings {
  void setSugar(int q) { ... }
  void setMilk(int q) { ... }
  void setChocolateDark() { ... }
  void setChocolateWhite() { ... }
}
class Chocolate {
  void fill(int q) { ... }
  void make() { ... }
}
```

(b) After the application of all the deltas

Figure 3: Delta Applications

immediate. While deltas add, remove or modify class, methods or fields, records operations add, remove or modify fields. Following this parallel, we can see class bodies as records, where each field corresponds either to a method of the class (the label being the method's name and the value its code), or to a field of the class (the label being the field's name and the value its value). Programs can also be seen as records, where each field corresponds to a class or an interface, with the label being the class (resp. interface) name and the value being the body of the class (resp. of the interface).

Based on this encoding into records, delta application can be modelled as a function manipulating a record corresponding to the core program, and delta composition is simply function composition. Thus we can use a type system defined for records [18] to check the application and composition of deltas. This type system is based on the following ideas: records are typed with *row types* that store all the information about the structure of the records; record operations are typed with *functional types* that map a row type corresponding to the input record to an output row type. The most important feature of this type system is *row polymorphism*, which gives types of record operations in terms of both *type* and *row variables*. These work together to model the fact that an operation can be applied to records with different structure, for example, capturing that it is possible to remove the field a from any record that has this field. In addition, these variables allow the types of the input of the operations to be related to their output, and thus allows the precise specification of the behavior of each operation.

We illustrate row polymorphism applied to delta-oriented programming with some examples. Figure 4 presents the type of the core program described in the previous section. Row types are delimited by curly brackets and consist of a finite sequence of field declaration followed by information about the remainder of the record. Field declarations are identified by a label, such as `Coffee` or `fill`, and state either that the field is present in the record, in which case the label is of the form **Pre**(τ), where τ is the type of the value contained in the field, or that the field is absent from the record, in which case the label is **Abs**. The additional row information can either be **Abs**, meaning that there are no more fields in the record, or a *row variable* ρ, meaning that the rest of the record is unknown. Hence, as our core program has the two classes `Coffee` and `TomatoJuice`, the type in Figure 4 states that only the two field `Coffee` and `TomatoJuice` are present, and both contain records corresponding to the bodies of the classes.

```
{
  Coffee : Pre( {
    fill : Pre;
    make : Pre;
    Abs
  } );
  TomatoJuice : Pre( {
    fill : Pre;
    make : Pre;
    Abs
  } );
  Abs
}
```

Figure 4: Core Program Type

Figure 5 presents the type of delta `D1`. Generally, the type of a delta is a *polymorphic* function type that is structured in three parts: the declaration of the type variables used in the type, the row type describing the input of the delta, and the output type of the delta. Typically, deltas have a row variable to describe the part of the program *not* affected by the delta. From Figure 5 we see that the type of the delta `D1` is polymorphic and uses the row variable ρ. The input type states that `D1` can take in input any program that does not have class `Settings`—the field `Settings` must be absent from the program—while the rest of it is undefined, and thus can be anything. The output type captures the modifications `D1` makes to the input program. The definition of field `Settings` has changed; it is now present, stating that class `Settings` has been added to the program, and that it contains two methods, `setSugar` and `setMilk`. As variable ρ is not changed, all other classes in the input program are not modified by `D1`.

```
∀ρ. { Settings : Abs; ρ }
 → { Settings : Pre ( { setSugar : Pre;
                        setMilk : Pre; Abs } ); ρ }
```

Figure 5: Delta `D1` Type

3.2 Syntax

$$
\begin{array}{rcl}
TP & ::= & \{CL^{\emptyset}\} \\
CL^c & ::= & \mathbf{Abs}^c \mid \rho^c \mid \mathtt{c} : CP;\ CL^{\{\mathtt{c}\}\uplus c} \\
CP & ::= & \mathbf{Abs} \mid \mathbf{Pre}(TC) \\
TC & ::= & \{FL^{\emptyset}\} \\
FL^f & ::= & \mathbf{Abs}^f \mid \rho^f \mid \mathtt{f} : FP;\ FL^{\{\mathtt{f}\}\uplus f} \\
FP & ::= & \mathbf{Abs} \mid \mathbf{Pre}
\end{array}
$$

Figure 6: Syntax of Program Types

The syntax of types for programs is presented in Figure 6. The type of a program, denoted TP, consists of a list (i.e., a row) of class declaration $CL^{\emptyset}$ enclosed by curly brackets. A row CL^c is annotated with a finite set of class names c representing the classes that cannot appear in CL^c. This restriction is to ensure that each class is declared only once.

A class list CL^c is either empty $\mathbf{Abs}^c$, capturing that classes whose names are not in c not present in the program; a variable ρ^c, meaning that the rest of the program, namely, the classes not in c, is unknown; or a declaration $\mathtt{c} : CP$ of class `c` followed by the declaration of the rest of the program $CL^{\{\mathtt{c}\}\uplus c}$. Note that $\uplus$ denotes disjoint union and its use ensures that `c` will not be declared again later. A class can either be absent (**Abs**) or present ($\mathbf{Pre}(TC)$), in which case the body of the class is described as a list TC (i.e., another row) of field and method types.

Analogously, row FL^f is annotated with a finite set of field and method names f that cannot appear again in FL^f. Such a row can either be empty ($\mathbf{Abs}^f$), unknown (ρ^f) or a declaration of a field (or method) `f` followed with the declaration of the rest of the class. A field (or method) can either be absent (**Abs**) or present (**Pre**) in the class. In the examples, when it is clear from the context, we shall omit the annotation on rows.

$$
\begin{array}{rcl@{\quad}l}
TD & ::= & TP \to TP \mid \forall\rho.TD & \text{Delta type} \\
TO & ::= & TC \to TC \mid \forall\rho.TO & \text{Operator type}
\end{array}
$$

Figure 7: Syntax of Delta Types

Figure 7 presents the syntax of types for deltas. As mentioned before, a delta type is a simple functional type $TP_1 \to TP_2$ describing the expected input (resp. the expected output) of the delta in TP_1 (resp. in TP_2). It is moreover possible for the delta type to be polymorphic in row variables ρ using the construct $\forall\rho.TD$. Finally, *operator types* TO are used to type the modifications specified in a **modifies class** C statement. TO follows exactly the same pattern as TP, except that their input and output type are TC, as they type class modifications.

3.3 Typing

This section applies the row polymorphic type system to the simple delta-oriented programming language presented Figure 8. This syntax covers all the important characteristics of applying deltas on products.

$$
\begin{array}{rcl}
PL & ::= & \mathbf{delta}\ \mathtt{d}\ D\ \ PL \mid P \\
P & ::= & \mathbf{core}\ \mathtt{C}\ \{[\mathbf{class}\ \mathtt{c}\ C]^*\} \mid \mathtt{d}(P) \\
C & ::= & \{\ F\ [;\ F]^*\ \} \\
F & ::= & T\ \mathtt{f}\ \mathit{def} \\
D & ::= & \{BD\ [;\ BD]^*\} \\
BD & ::= & \mathbf{adds\ class}\ \mathtt{c}\ C \mid \mathbf{removes\ class}\ \mathtt{c} \\
 & \mid & \mathbf{modifies\ class}\ \mathtt{c}\ O \\
O & ::= & \{BO\ [;\ BO]^*\} \\
BO & ::= & \mathbf{adds}\ F \mid \mathbf{removes}\ \mathtt{f}
\end{array}
$$

Figure 8: Syntax of a simple DOP Language

A program PL is structured in two parts: PL includes all the deltas **delta** `d` D used in the product line; and in P these deltas are applied in order to a core product (i.e., a set of classes) to generate a product. A class **class** `c` C is identified with a name `c` and contains a non-empty set of fields or methods separated by a semicolon. Fields and methods F have a (return) type T, are identified by a name `f` and have a definition that is empty for a field or is a list of arguments and a method body. A delta **delta** `d` D has a name `d` and some code D consisting of a list of delta operations DB. The three basic delta operations are: **adds** C adds a new class; **removes** `c` removes a class; and **modifies** `c` O modifies a class, where O is a list of operations that remove or add fields.

Applying row polymorphism to such a simple calculus produced a type system quite close to the original one for general records. Indeed, most of the typing rules used in our type system exist already in the original work on rows; we simply instantiated them for DOP. For instance, the classical typing rule for records is used to type classes and programs; the addition and removal operations on classes and fields are analogous to the same operators on records; composition and application of operations are typed like classical function composition and application. Compared to Remy [18], we added the type for class modification (which Remy encoded as a composition of field access, deletion and addition [18]). The other difference with classic type systems for functional programming is only technical and concerns the type generalization which, in our case, is totally free. No additional restrictions are required on type generalization, as our calculus does not have λ-abstraction.

We present our type system in three steps. First, we present core product typing using instantiations of the typing rule for records. Then, we present delta typing using instantiations of the typing rules for record modifications and function composition. Finally, we present how we type a global product line using the previous rules and rules analogous to those for `let` in ML [19] and for function application.

Typing Core Products.

The type rules for a core product, i.e., a set of classes, are presented Figure 9.

The rule TClass is used to type the body of a class, i.e. a set of field and method declarations. This rule takes all the fields and method defined in the class body and simply states in the output type that these elements are present in the class body, forgetting their types and definitions. The rule TProgram is used to type a set of classes. It first

(TClass)
$\{T_1\ \mathtt{f}_1\ def_1; \ldots; T_n\ \mathtt{f}_n\ def_n\} : \{\mathtt{f}_1 : \mathbf{Pre}; \ldots; \mathtt{f}_n : \mathbf{Pre}; \mathbf{Abs}\}$

(TProgram)
$$\frac{C_i : TC_i \quad i \in 1..n}{\begin{array}{l}\mathtt{core\ C\ \{class\ c}_1\ C_1\ \ldots\ \mathtt{class\ c}_n\ C_n\mathtt{\}} \\ : \{\mathtt{c}_1 : \mathbf{Pre}(TC_1); \ldots; \mathtt{c}_n : \mathbf{Pre}(TC_n); \mathbf{Abs}\}\end{array}}$$

Figure 9: Typing Core Products

types the bodies of all the classes using the previous rule, and uses the resulting types to construct the type of the whole program containing all the classes.

Typing Deltas.

The typing rules for deltas are presented in Figure 10. Typing of deltas is a little more complex than typing core products, as deltas are essentially polymorphic functions.

(TInst) $\dfrac{E : \forall\rho.T}{E : T}$ (TGen) $\dfrac{E : T}{E : \forall\rho.T}$ (TSubst) $\dfrac{E : T}{E : \sigma(T)}$

(TSeq)
$$\frac{E : T_1 \to T_2 \quad E' : T_2 \to T_3}{E; E' : T_1 \to T_3}$$

(TAddField)
$\mathbf{adds}\ T\ \mathtt{f}\ def : \forall\rho.\{\mathtt{f} : \mathbf{Abs}; \rho\} \to \{\mathtt{f} : \mathbf{Pre}; \rho\}$

(TRemoveField)
$\mathbf{removes}\ T\ \mathtt{f}\ def : \forall\rho.\{\mathtt{f} : \mathbf{Pre}; \rho\} \to \{\mathtt{f} : \mathbf{Abs}; \rho\}$

(TAddClass)
$$\frac{C : TC}{\mathbf{adds\ class}\ \mathtt{c}\ C : \forall\rho.\{\mathtt{c} : \mathbf{Abs}; \rho\} \to \{\mathtt{c} : \mathbf{Pre}(TC); \rho\}}$$

(TRemoveClass)
$\mathbf{removes\ class}\ \mathtt{c} : \forall\rho, \rho'.\{\mathtt{c} : \mathbf{Pre}(\{\rho\}); \rho'\} \to \{\mathtt{c} : \mathbf{Abs}; \rho'\}$

(TModifiesClass)
$$\frac{O : TC_1 \to TC_2 \quad \rho \text{ fresh}}{\begin{array}{l}\mathbf{modifies\ class}\ \mathtt{c}\ O \\ : \forall\rho.\{\mathtt{c} : \mathbf{Pre}(TC_1); \rho\} \to \{\mathtt{c} : \mathbf{Pre}(TC_2); \rho\}\end{array}}$$

Figure 10: Typing Deltas

The typing rules deal with three different concerns. Firstly, there are four rules that deal with polymorphism and sequential composition of functions. Secondly, two rules give types to the basic operators on class bodies; and finally, three rules type the basic operators on program structure.

To simplify the presentation of our type system and avoid duplication of typing rules, in the first category we use the symbol E to denote either the code of a delta D, a simple delta operation BD, the code of a class body modification O, a simple function BO or a delta name `d`. Moreover, we use the symbol T to denote either a class body type TC, a product type TP, a type TO of a class body manipulation TO or a type TD of the code of a delta.

The rules TInst and TGen simply state that we can bind and unbind type variables at will: indeed we only use variable name bindings to avoid unexpected variable name capture. The rule TSubst makes use of classic *substitutions* σ to replace row variables with more concrete types, thus allowing the input and output types of a function to be made more precise. Finally, the rule TSeq types the sequential composition of two functions E and E'. Note that this rule enforces that the output of E is a valid input of E' by requiring that the output type of E is the same as the input type of E'—typically achieved using TSubst.

The basic operations modifying the class structure are typed with the two rules TAddField (for field and method addition) and TRemoveField (for field and method removal). The input type of the **adds** operator specifies that the added field (or method) should not be present in the input class (denoted by `f:`**Abs**), while the rest of the class is unconstrained (denoted by row variable ρ), and the output type of the operator captures the modification of the input class by specifying that the added field (or method) is present in the output. Dually, the type for the removal operator specifies that the input class must contain the field (or method) to be removed, and that the output is identical to the input with the removed field.

Finally, the basic operation to add, remove and modify classes from a product are typed using rules TRemoveClass (for class removal), TAddClass (for class addition) and TModifyClass (for class modification). Analogous to the typing rule TRemoveField, the rule TRemoveClass specifies that the input product must contain the class to be removed (denoted by `c:`**Pre**`({`ρ`})`) and results in the same product with the specified class removed. Rule TAddClass first computes the type of the class to add, and from this generates the type of the addition operator. As for rule TAddField, this rule specifies that the input product must not already have the class to be added (denoted by `c:`**Abs**) and results in the same product extended with the new class. To type the modification of a class, we first compute the type of the modification with the statement $O : TC_1 \to TC_2$; this modification takes a class body of type TC_1 and produces a class body typed TC_2. Then, the overall modification of the class `c` expects as input a product with the class `c` present with body typed TC_1. Its output type states that the product is left unchanged except for `c`, whose body is now typed TC_2.

Typing Product Lines.

The typing judgements for product lines have the form $\Gamma \vdash E : T$, where Γ is the typing environment storing the type of deltas; E is a term and T its type in the context of Γ. The type rules for product lines are presented in Figure 11.

The rule TDelta types delta declarations. A delta **delta** `d` D is typed by first typing its code D, then we continue the typing of the rest PL of the product line with an extended typing environment. The rule TProduct is used to type the core product of the product line. The rule TDName is used to type the name of a delta, which appears when we apply a delta to a product. This is done simply by looking in the environment Γ. Finally, the rule TApp types the application

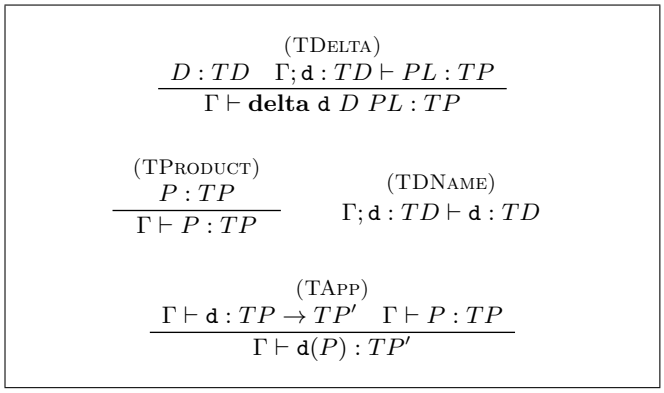

Figure 11: Typing Product Lines

of a delta `d` to a product P. This rule simply computes the types of `d`, of P, ensures that P is a valid input for `d`, and defines the type of the resulting product as the output type of the delta `d`.

Properties.

The presented type system can be used to check that a particular application of deltas on a core product is *structurally* consistent, i.e., the classes are added, removed and modified only when such operations are possible. This is done by considering only the structure of the program, tracking each modification and checking its validity using a row-based type system. The following theorem states that our type system indeed ensures the consistency of delta applications:

Theorem 1. *Given a product line PL such that there exist a type TP and derivation of $\emptyset \vdash PL : TP$. Then PL is structurally consistent.*

3.4 Example

To illustrate our type system, we present two typing examples. First, we show how to validate the application of `D1` and `D2` (Figure 2) to the core program of Section 2 (Figure 1). Then we demonstrate that applying `D3` on the same core program would fail.

Successful Application.

Let's follow the typing rules to see how to validate the product line composed by applying two deltas `D1` and `D2` to the core program `P` (Figure 1), namely, `D2(D1(P))`. Using the typing rules TCLASS and TPROGRAM it is easy to see that `P` is well-typed, as presented before in Figure 4. Denote this type by TP. Using rule TCLASS, to type the class `Settings`, and TADDCLASS, we can also see that the delta `D1` is also well-typed, as presented before in Figure 5. Denote this type by TD_1. Using the same approach, we can see that the delta `D2` can be typed with TD_2, defined as follow:

```
TD2 = ∀ρt, ρ. { TomatoJuice : Pre({ρt}); ρ }
          → { TomatoJuice : Abs; ρ }
```

Now, using a substitution σ that replaces the variable ρ with the content of the type of P, we can instantiate the type of delta `D1` so that it can take program P as a parameter, resulting in the following type:

```
{Settings: Abs;
 Coffee: Pre({fill: Pre; make: Pre; Abs});
 TomatoJuice: Pre({fill: Pre; make: Pre; Abs});
 Abs } →
{Settings: Pre({setSugar: Pre;
                setMilk: Pre; Abs});
 Coffee: Pre({fill: Pre; make: Pre; Abs});
 TomatoJuice: Pre({fill: Pre; make: Pre; Abs});
 Abs }
```

Using typing rule TAPP, we can easy check that `D1(P)` is typable; its type is the output type of the type just presented.

To type the application of `D2` to the product `D1(P)`, we can apply the same technique of instantiating TD_2 so that its input type matches the type of `D1(P)`. The output type of the result, i.e. the type of `D2(D1(P))`, is:

```
{Settings: Pre({setSugar: Pre;
                setMilk: Pre; Abs});
 Coffee: Pre({fill: Pre; make: Pre; Abs});
 Abs }.
```

Failing Application.

Let's now consider what happens when applying `D3` (Figure 2) to program `P`. To check the application, we first need to find the type of `D3`. This delta consists of two operations: adding class `Chocolate` and modifying class `Settings`. Using the typing rules TCLASS, TADDCLASS, TADDFIELD, TMODIFYCLASS and TSEQ, `D3` can be typed with TD_3 as follows:

```
TD3 = ∀ρt, ρ.
 { Chocolate: Abs;
   Settings: Pre({setChocolateDark: Abs;
                  setChocolateWhite: Abs; ρt});
   ρ } →
 { Chocolate: Pre({fill: Pre; make: Pre; Abs});
   Settings: Pre({setChocolateDark: Pre;
                  setChocolateWhite: Pre; ρt});
   ρ }
```

To be able to type the application of `D3` to `P`, we must instantiate TD_3 so that its input type corresponds to TP, the type of P. But this is impossible, because the class `Settings` is declared as being absent in TP, while it is required to be present in the input type of TD_3. Hence typing fails.

4. DISCUSSION AND LIMITATIONS

As shown in the examples, our type system can naturally catch errors in delta application and delta composition. Nevertheless, this type system is still incomplete and unable to check errors in general DOP. The first element that is missing from the current approach is the ability to check the type safety of the computed product. Indeed, it is possible that after error-free delta application that the resulting product is missing classes it needs to execute. The second missing element is the ability to type complete product lines dealing with the different products generated depending on which features are selected. The straightforward approach of typing every generated product separately is unsatisfactory as it requires checking a number of products exponentially larger than the number of deltas.

5. RELATED WORK

The goal of type checking the code base of a software product line is to ensure that the generated products are type safe, up to the degree of type safety provided by the base language, *without* having to actually generate the products.

Other static analysis techniques can instead be employed to check for other potential deficiencies, without aiming to be ensure complete type safety.

Thaker et al. [23] describe an informally specified approach to the safe composition of software product lines that guarantees that no reference to an undefined class, method or variable will occur in the resulting products. The approach is presented modulo variability given in the feature model and deals especially with the resulting combinatorics. The lack of a comprehensive formal model of the underlying language and type system was rectified with *Lightweight Feature Java* (LFJ) [8]. Underlying LFJ is a constraint-based type system whose constraints describe composition order, the uniqueness of fields and methods, the presence of field and methods along with their types, and feature model dependencies. The soundness of LFJ's type system was validated using theorem prover Coq.

Featherweight Feature Java (FFJ) [3], a formal model of a feature-oriented Java-like language, also formalises Thaker et al.'s [23] approach to safe composition, although for this system type checking occurs only on the generated product. *Coloured Featherweight Java* [12], which employs a notion of colouring of code analogous to but more advanced than `#ifdef`s, lifts type checking from individual products to the level of the product line and guarantees that all generated products are type safe. More recent work [2] refines the work on FFJ, expressing code refinements as modules rather than low-level annotations. The resulting type system again works at the level of the product line and enjoys soundness and completeness results, namely, that a product line is well-typed if and only if all of its derived products are well-typed.

The formal models just described are all extensions of Featherweight Java [11], which is a rather impoverished base language. Apel and Hutchins [1] propose *gDeep* as a possible unifying foundation for languages for feature-oriented programming. While not dealing with feature models and variability directly, gDeep does have a more advanced language core than the formalisms based on Featherweight Java, and thus offers a stepping stone for more advanced programming language constructs to be used in formal models of programming languages and type systems for software product lines.

In the above mentioned work the refinement mechanisms are monotonic, so no method/class removal or renaming is possible. Kuhlemann et al. [15] addresses the problem of non-monotonic refinements, though their approach does not consider type safety. They consider the presence of desired attributes depending upon which features are selected. Checking is implemented as an encoding into propositional formulas, which are fed into a SAT solver. Recent work addresses non-monotonic refinement mechanisms that can remove or rename classes and methods. An alternative approach due to Schaefer *et al.* [20] generate detailed dependency constraints for checking delta-oriented software product lines. The checking of the constraints is performed per product, rather than at the level of product lines. This approach to typing delta-oriented programs is complementary to our work, providing part of the checking we have omitted.

A number of static analysis techniques have been developed for the design models or code of software product lines. Heidenreich [10] describes techniques for ensuring the correspondence between feature models, solution-space models, and problem-space models, which is realised in the FeatureMapper tool. In this tool, models are checked for well-formedness against their meta-model. Similarly, Czarnecki and Pietroszek [7] provide techniques for ensuring that no ill-structured instance of a feature-based model template will be generated from a correct configuration. Apel et al. [4] present a general, language independent, static analysis framework for reference checking—checking which dependencies are present and satisfied. This is one of the key tasks of type checking a software product line. Similar ideas are applied in a language-independent framework for ensuring the syntactic correctness of all product line variants by checking only the product line itself, again without having to generate all the variants [13]. Clarke et al. [5] present an abstract framework for describing about conflicts between code refinements and conflict resolution in the setting of an abstract version of delta-oriented programming. Padmanabhan and Lutz [17] describe the DECIMAL tool, which performs a large variety of consistency checks on software product line requirements specifications, in particular, when a new feature is added to an existing system. Techniques developed for the analysis and resolution of interference of aspects in AOP [14, 9] address similar problems to analyses of software product line conflicts, but they do not consider variability.

6. CONCLUSION

This paper presented a type system for delta-oriented programming based on row polymorphism, which is traditionally used to type extensible records. The novelties of this paper are the application of this type system in a totally different domain, and the demonstration that it fits perfectly to ensure the consistency of delta application and delta composition for the construction of software product lines.

The obvious shortcoming of our approach is that the consistency of the products, as well as modularly checking the entire product line, are not yet considered. This was done on purpose, as we wanted to highlight the connection between deltas and rows, but also because we want to build our type checking approach incrementally, and avoid the complexity of the global approach of, for example, Schaefer et al. [20].

Our next step will be to extend our type system to enable classic object-oriented typing of the generated product, without having to generate it. From there on we will endeavour to build a simple, modular, and efficient type system for delta-oriented product lines.

7. REFERENCES

[1] Sven Apel and DeLesley Hutchins. A calculus for uniform feature composition. *ACM Trans. Program. Lang. Syst.*, 32(5), 2010.

[2] Sven Apel, Christian Kästner, Armin Größlinger, and Christian Lengauer. Type safety for feature-oriented product lines. *Autom. Softw. Eng.*, 17(3):251–300, 2010.

[3] Sven Apel, Christian Kästner, and Christian Lengauer. Feature Featherweight Java: A calculus for feature-oriented programming and stepwise refinement. In Yannis Smaragdakis and Jeremy G. Siek, editors, *GPCE*, pages 101–112. ACM, 2008.

[4] Sven Apel, Wolfgang Scholz, Christian Lengauer, and Christian Kästner. Language-independent reference checking in software product lines. In *Proceedings of the 2nd International Workshop on Feature-Oriented*

Software Development, FOSD '10, pages 65–71, New York, NY, USA, 2010. ACM.

[5] Dave Clarke, Michiel Helvensteijn, and Ina Schaefer. Abstract delta modeling. In *Proceedings of the ninth international conference on Generative programming and component engineering*, GPCE '10, pages 13–22, New York, NY, USA, 2010. ACM.

[6] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[7] Krzysztof Czarnecki and Michal Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In Robert Glück and Michael R. Lowry, editors, *GPCE*, volume 3676 of *Lecture Notes in Computer Science*, pages 422–437. Springer, 2005.

[8] Benjamin Delaware, William R. Cook, and Don S. Batory. Fitting the pieces together: a machine-checked model of safe composition. In Hans van Vliet and Valérie Issarny, editors, *ESEC/SIGSOFT FSE*, pages 243–252. ACM, 2009.

[9] Rémi Douence, Pascal Fradet, and Mario Südholt. A framework for the detection and resolution of aspect interactions. In Don S. Batory, Charles Consel, and Walid Taha, editors, *GPCE*, volume 2487 of *Lecture Notes in Computer Science*, pages 173–188. Springer, 2002.

[10] Florian Heidenreich. Towards systematic ensuring well-formedness of software product lines. In *Proceedings of the 1st Workshop on Feature-Oriented Software Development*, pages 69–74, New York, NY, USA, oct 2009. ACM.

[11] 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, 2001.

[12] Christian Kästner and Sven Apel. Type-checking software product lines - a formal approach. In *ASE*, pages 258–267. IEEE, 2008.

[13] Christian Kästner, Sven Apel, Salvador Trujillo, Martin Kuhlemann, and Don S. Batory. Guaranteeing syntactic correctness for all product line variants: A language-independent approach. In Manuel Oriol and Bertrand Meyer, editors, *TOOLS (47)*, volume 33 of *Lecture Notes in Business Information Processing*, pages 175–194. Springer, 2009.

[14] Emilia Katz and Shmuel Katz. Incremental analysis of interference among aspects. In Curtis Clifton, editor, *FOAL*, pages 29–38. ACM, 2008.

[15] Martin Kuhlemann, Don S. Batory, and Christian Kästner. Safe composition of non-monotonic features. In Jeremy G. Siek and Bernd Fischer, editors, *GPCE*, pages 177–186. ACM, 2009.

[16] Kwanwoo Lee, Kyo Chul Kang, and Jaejoon Lee. Concepts and guidelines of feature modeling for product line software engineering. In *Proceedings of the 7th International Conference on Software Reuse: Methods, Techniques, and Tools*, ICSR-7, pages 62–77, London, UK, UK, 2002. Springer-Verlag.

[17] Prasanna Padmanabhan and Robyn R. Lutz. Tool-supported verification of product line requirements. *Autom. Softw. Eng.*, 12(4):447–465, 2005.

[18] Didier Rémy. *Type inference for records in natural extension of ML*, pages 67–95. MIT Press, Cambridge, MA, USA, 1994.

[19] Didier Rémy. Using, understanding, and unraveling the ocaml language. from practice to theory and vice versa. In Gilles Barthe, Peter Dybjer, Luís Pinto, and João Saraiva, editors, *Applied Semantics*, volume 2395 of *Lecture Notes in Computer Science*, pages 115–137. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-45699-6_9.

[20] Ina Schaefer, Lorenzo Bettini, and Ferruccio Damiani. Compositional type-checking for delta-oriented programming. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD '11, pages 43–56, New York, NY, USA, 2011. ACM.

[21] Ina Schaefer, Lorenzo Bettini, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In *Proceedings of the 14th international conference on Software product lines: going beyond*, SPLC'10, pages 77–91, Berlin, Heidelberg, 2010. Springer-Verlag.

[22] 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.

[23] Sahil Thaker, Don S. Batory, David Kitchin, and William R. Cook. Safe composition of product lines. In Charles Consel and Julia L. Lawall, editors, *GPCE*, pages 95–104. ACM, 2007.