Compositional Type-Checking of Delta-oriented Product Lines

Ina Schaefer†  Lorenzo Bettini  Ferruccio Damiani

Dept. of Computer Science and Engineering  Dipartimento di Informatica
Chalmers University of Technology  Università di Torino
421 96 Gothenburg, Sweden  I-10149 Torino, Italy

Abstract

Delta-oriented programming (DOP) is a compositional approach to flexibly implement software product lines. A delta-oriented product line is realized by a code base and a product line declaration. The code base consists of delta modules comprising modifications of object-oriented programs. The product line declaration defines how these modifications have to be combined to generate all possible products. This talk presents a foundation for compositional type checking of delta-oriented product lines of JAVA programs using a minimal core calculus for DOP. The calculus is equipped with a constraint-based type system that allows analyzing each delta module in isolation, so that the results of the analysis can be reused (like the delta module themselves) across different product lines. By combining the results of the delta module analysis with the product line declaration, it can be established whether all the products of the product line are well-typed according to the JAVA type system.

1 Background

A software product line (SPL) is a set of software systems with well-defined commonalities and variabilities [3]. Delta-oriented programming (DOP) [11][12] is a flexible compositional approach for implementing software product lines. It relies on the notion of program deltas [13][10], a term that was first used in [9] to describe the modifications of object-oriented programs. The implementation of a delta-oriented product line consists of a code base and a product line declaration. The code base comprises a set of delta modules describing modifications of object-oriented programs that are necessary to generate all possible product implementations. A delta module can add classes, remove classes or modify classes by changing the class structure. The product line declaration provides the connection between the delta modules and the product members of a product line that are usually determined in terms of product features [6]. In the product declaration, it is specified for which feature configurations a delta module has to be used and in which order the delta modules that are applicable for a particular feature configuration are applied to generate the corresponding product. Separating the application conditions and delta module ordering from the definition of the delta modules increases the reusability of the delta modules, making it possible to develop different product lines by sharing delta modules.

Delta-oriented programming is an extension of feature-oriented programming (FOP) [2], a prominent compositional approach for implementing software product lines, by explicit operations to remove classes, methods or fields from a program. In [12], we have shown that FOP can indeed be embedded straightforwardly into DOP. While in FOP a product line implementation always starts from base feature modules comprising common core functionalities, in DOP, any product can be chosen as starting point of product generation. Hence, DOP supports proactive product line development, where all products are planned in advance, as well as proactive and extractive product line development [8], where development starts from an initial product line or existing legacy products. Moreover, the application conditions associated to the delta modules allow handling combinations of features explicitly providing a way to counter the optional feature problem [7].

*Partially supported by the German-Italian University Centre (Vigoni program) and by MIUR (PRIN 2009 DISCO).
†This author has been supported by the Deutsche Forschungsgemeinschaft (DFG) and the EU project FP7-231620 HATS.
2 Motivation

The flexibility of DOP to start product generation from any complete or partial product as well as the expressiveness to handle feature combinations explicitly makes it challenging to ensure that for every valid feature configuration a unique product can be generated and that the SPL is type safe. A SPL is called type safe if all possible products are well-typed programs according to the type system of the programming language used to implement the products. In principle, the type safety of a SPL could be checked by generating and type-checking all products separately. A major drawback of this approach is that it is hard for the programmer to understand which delta module causes an error, based on the results while type-checking a single product.

Therefore, a first requirement for a type system for delta-oriented product lines is that it allows ensuring the type safety of the SPL without generating and inspecting the code of all possible products. The separation of application conditions and delta module ordering from the definition of the delta modules allows reusing delta modules across different product lines. Hence, a second requirement for the design of a type system for DOP is that each delta module can be analyzed in isolation without relying on global information of the product line.

3 Contribution

In this work, we address the above requirements by developing IMPERATIVE FEATHERWEIGHT DELTA JAVA (IFAJ) a core calculus for DOP of product lines of JAVA programs, based on IFJ (an imperative variant of FJ [5]) that is used as underlying programming language for the products. An IFJ program consists of a class table CT, that is, a mapping from class names to class definitions.

As first step towards a compositional type system for DOP, we define a constraint-based type system for IFJ that infers a set of class constraints £ for an IFJ program CT. These constraints can be checked against the class signature table of CT (i.e., the program CT deprived from method bodies) in order to establish whether CT is a well-typed IFJ program. The pair ⟨signature(CT), £⟩ represents an abstract representation of the program CT that can be used to establish whether CT is type safe.

In the second step, we define an abstract representation of the delta modules in IFAJ, that can be inferred by analyzing each delta module in isolation and allows deriving abstract representations of the possible products directly without generating and inspecting their code (first requirement). The abstraction of a delta module δ consists of a pair ⟨signature(δ), Dδ⟩, where signature(δ) is the delta module signature and Dδ is a set of delta clause-constraints. The signature of a delta module δ is the analogue of a class signature for a delta module, i.e., a representation of the delta module without method bodies. It can be inferred by a straightforward inspection of the code of the delta module. The delta clause-constraints are inferred by a constraint-based type system for IFAJ that analyzes each delta module δ in isolation.

A difficulty to ensure type safety by considering only the code of the delta modules is that delta modules are incomplete and only define differences to existing products. Thus, a type system for delta-oriented product lines that only requires to inspect the code of each delta module in isolation (second requirement) has to explicitly capture the expectations a delta module has on the context in which it is applied. The delta clause-constraints express exactly these expectations of a delta module during delta module application and can be reused across different product lines, as they only depend on the delta module itself. The inferred sets of delta clause-constraints are organized such that the set of class constraints required to establish type safety of a product can be derived directly without generating and inspecting the code of the product. Therefore, type safety of a product line can be established only from the delta module signatures, the delta clause-constraints and the product line declaration.
Checking type safety of a DOP product line is linear in the number of its products (which may be exponential in the number of features). We expect that generating the constraints and performing the associated checks will be more efficient than generating all products and checking them by a JAVA compiler. Moreover, in the latter case it would be hard for the programmer to determine which delta module causes an error from the result of the JAVA compiler. Instead, the constraints collected by the IFAJ type system can be exploited to show which part of a delta module generated the error. The idea (not formalized in current presentation of the type system) is to keep track of the location of the code in a delta module for each generated constraint.

4 Related Work

The calculus LIGHTWEIGHT FEATURE JAVA (LFJ) [4], based on LJ (LIGHTWEIGHT JAVA) [14], provides a formalization of FOP [2] together with a constraint-based type system that satisfies the two design requirements described in Section 2. The calculus FEATHERWEIGHT FEATURE JAVA for Product Lines (FFJPL) [1], based on FJ, comprises an (independently developed) type system for FOP that does not meet the second requirement. These approaches for ensuring type safety of feature-oriented product lines are not straightforwardly adaptable to deal with the additional flexibility provided by DOP.

References