Synthesizing Type-safe Compositions in Feature Oriented Software Designs using Staged Composition

Boris Düdder, Jakob Rehof
Technical University of Dortmund
Otto-Hahn Str. 12
Dortmund, Germany
boris.duedder@tu-dortmund.de
jakob.rehof@tu-dortmund.de

George T. Heineman
Worcester Polytechnic Institute
100 Institute Road
Worcester, MA, USA
heineman@cs.wpi.edu

ABSTRACT

The composition of features that interact with each other is challenging. Algebraic formalisms have been proposed by various authors to describe feature compositions and their interactions. The intention of feature compositions is the composition of fragments of documents of any kind to a product that fulfills users' requirements expressed by a feature selection. These modules often include code modules of typed programming languages whereas the proposed algebraic formalism is agnostic to types. This situation can lead to product code which is not type correct. In addition, types can carry semantic information on a program or module. We present a type system and connect it to an algebraic formalism thereby allowing automatic synthesis of feature compositions yielding well-typed programs.

Categories and Subject Descriptors

I.2.2 [**ARTIFICIAL INTELLIGENCE**]: Automatic Programming—*Program synthesis*; F.4.1 [**MATHEMATICAL LOGIC AND FORMAL LANGUAGES**]: Mathematical Logic—*Logic and constraint programming*

Keywords

Feature composition, automatic program synthesis, type theory, combinatory logic

1. INTRODUCTION

Features and aspects are an established way to design software systems. Features are an important concept in software product line research, and feature-oriented programming concepts have been explored.

Features are extensions of program functionality and directly reflect users' requirements. A composition of features is a software product. In order to synthesize such products, it is interesting to analyze a way to automatically compose features into software products, respectively, their implementation code. Because features are not formally linked to types, such compositions might be valid feature compositions but might lead to uncompilable implementation code due to typing errors. In this paper a type system is proposed which types features and their modules, thereby enabling automatic synthesis of type-safe feature- and module compositions. A feature composition goal specifies set of feature selections that correspond to product line members.

The complication of composing features that include feature interactions is discussed for example by Liu, Batory, and Nedunuri [1] and Prehofer [2]. We follow their motivation and extend their intention towards automatic feature composition.

An algorithm for combinatory logic synthesis [3, 4] is used to synthesize type-safe compositions of features wrt. to a given target specification obeying the constraints posed by individual feature interactions. The synthesis algorithm is based on the type inhabitation problem that is the decision problem: given a type environment Γ representing a feature model and a type τ representing a feature composition goal, can we generate a valid composition specification e, a feature selection, that is composable from modules and features in Γ and satisfies the selection τ, noted as $\Gamma \vdash e : \tau$? Here, types are more than simple data-types in implementation languages. Such types can be augmented with semantic information, e.g. of name or properties of a features. As soon as a feature model, modules and code generators for each single feature are given, this leads to an automated process, directly presenting the user with all valid and ready to execute products on the input of her individual feature requirements. Staged composition synthesis [5, 6] allows such an automated process and guarantees the generation of well-typed product code. A program with a given type τ satisfies a valid feature composition that is determined by τ.

The paper is structured as follows. Features and their interactions in composition are discussed in Section 3, and a formal notion of feature composition in Section 4. A type system capable of expressing semantic properties of feature compositions is introduced in Section 5. Section 6 combines the notion of feature composition with type components for synthesizing automatically type-safe feature compositions resulting in well-typed implementation code. In Section 2 we discuss related work. Section 7 concludes the paper.

The contributions of this paper are: joining feature compositions and a suitable type system, a transformation of typed features and modules into a representation suitable

SPLC 2015, July 20 - 24, 2015, Nashville, TN, USA

DOI: http://dx.doi.org/10.1145/2791060.2793677

for automatic program synthesis. We follow the approach of Batory [1] and Prehofer [2] by expressing features and their composition by algebraic relations.

2. RELATED WORK

The book on Feature-Oriented Software Product Lines [7] by Apel, Batory, Kästner ans Saake provides a detailed overview and useful starting point on the subject. A broader overview is given by Apel in [8]. For a specific review of the automated analysis and formal treatment of feature models we refer the interested reader to [9]. The line of work we follow is mainly concerned with automatic synthesis of feature configurations. Starting with the insight that feature models can be represented as propositional formulas [10, 11, 12], various techniques have been employed to synthesize feature selections. A type-safe composition of product lines, using simple types of the implementation language, has been proposed in [13]. Intersection types are more powerful for specification [14]. Intersection type inhabitation has been used to synthesize mixin composition chains [15]. The further exploration of the role of staging [5] in type-inhabitation driven composition of object-oriented code [4] is an important goal for future research.

Liu, Batory, and Nedunuri [1] have proposed an algebraic formalism that is focused on the expression of feature and module compositions to respect feature interactions. Prehofer [2] develops this approach further and extends it to semantic refinement. We use the work of both groups for expressing feature and module compositions with semantic refinement and provide a type system for typing features and modules as well as introducing semantic types that allow for semantic refinements.

The line of work presented in [16, 17] also performs synthesis, but with the goal of extracting feature models from logic specifications. Via the connection between diagrams and grammars it might be possible to pipeline these approaches, in order to use a logic for feature specification as a starting point instead of a grammar.

3. FEATURE INTERACTIONS

We discuss feature interactions by means of an example given in [1, 2, 18]. Assume that we are given three Java classes implementing a stack of characters, an integer counter and a simple thread-safe lock.

```
class Stack {
 String s = new String();// Use Java Strings
 void empty() {s = ""; }
 void push(char a)
        {s = String.valueOf(a).concat(s); }
 void pop() {s = s.substring(1); }
 char top() { return (s.charAt(0) ); } }

class Counter {
 int i = 0;
 void reset() {i = 0; }
 void inc() {i = i+1; }
 void dec() {i = i-1; }
 int size() {return i; } }

class Lock {
 boolean locked = false;
 void lock() {locked = true;}
 void unlock() {locked = false;}
 boolean is_unlocked() {return ! locked;} }
```

These three classes are implementations of three features: Stack, Counter, and Lock. From these Java feature implementations we can derive AOP implementations that allow the composition of these three features, e.g. to a thread-safe stack with a counter. An adaptor of a refinement from Stack to Counter is needed to be woven into the Stack code when adding a Counter to a Stack. The number of adaptors grows linearly with the length of compositions and exponentially in the number of features in the worst-case. We will now consider an algebraic notation to describe features, modules, adaptors and *formalize* their compositions.

Our example can be seen as a special case, that the three features above are implemented as three separate classes. This is usually not the case and our method support various variation points similar to [19].

4. NOTION OF FEATURE COMPOSITION

We use an algebraic notation developed in [1] for the syntactic description of compositions and weavings in AOP and FOP. For adaptors, we use the notation from [1, 2]. This concepts has been further developed by Batory et. al. in [19] and is compatible with our approach. For every feature F, G, H there exists a corresponding base implementation f, g, h. In our examples we identify these base modules with classes but the notion is more general. An adaptor of h to a feature G is denoted as the derivative $\partial h/\partial G$ and means the refinement of module h to a feature G. An adaptor of feature X to feature Y adapts the methods of feature X to the context of feature Y. With this notation we capture the effects of features G to specific (class/code) modules h. Furthermore, we define the disjoint syntactic combination of code by $f + g$ as the symmetric aggregation of two modules.

The (non-symmetric) function composition or weaving of p and h is written as $p * h$ and means the composition of two derivatives or a derivative to a module. Code of a feature H is notated as $[H]$. Basic properties are distributivity $d*(a+b) = d*a+d*b$ and $\partial b/\partial H * a = a$ for two different base modules of two features. We can define $\partial/\partial F$ as derivative of F for any other modules. We define for a module m the application of $\partial/\partial F$ to m as $\partial/\partial F\, m$ or $\partial m/\partial F$. Distributivity of over $+$ and $*$ are defined as $\partial/\partial F\,(a+b) = \partial a/\partial F + \partial b/\partial F$ and $\partial/\partial F\,(a*b) = \partial a/\partial F * \partial b/\partial F$. We extend this notation to higher-order differentials, e.g. the second-order derivative $\partial^2 x/\partial H \partial G$ being the differential of module x wrt. to features H and G.

The composition of two features can be defined as: $[H(f)] = h + \partial f/\partial H * f$. In the following section, we introduce a type system that can be used to type feature composition operations presented in this section.

5. TYPED FEATURE COMPOSITION

We introduce an type system with intersection types [20] over types τ and σ defined by $\tau, \sigma ::= \alpha \mid s \mid \tau \to \sigma \mid \tau \cap \sigma \mid D(\tau, \ldots, \sigma) \mid \Box\tau$. The type variable α is drawn from a set of type variables ranging over implementation types such as Integer or String. Semantic types [6], ranged over by s, represent semantic properties of modules and features like lock, counter, or stack. Such semantic types allow to express semantic specifications for compositions of features as presented in [2]. Furthermore, Venneri's results [21] show that

we can also express conditional refinements of compositions, e.g. mentioned in [2]. A function type $\tau \to \sigma$ has domain type τ and range type σ. An expression has intersection $\tau \cap \sigma$ if and only if it has type τ and type σ. Type constructors are ranged over by D, e.g. *List* for constructing a list or *Pair* for constructing pairs. The modal type operator $\Box$ introduced in [5] allows to distinguish between code of type τ ($\Box\tau$), representing code that when executed results in (implementation) type τ, and expressions with type τ. The operator precedence is $\Box$ over $\cap$ over $\to$. This allows staging [5, 6] of computation: a meta-computation of type $\Box\tau$ is guaranteed to result in generated code of type τ.

Term expressions ranged over by e, e' etc. are purely applicative, defined by $e, e' ::= x \mid (e\, e')$. Term variables or combinators x name modules and derivatives, linking to their implementation. An expression of the form $(e\, e')$ denotes the application of operator e to argument e'. We assign type assumptions to modules h and features F by type assignments of the form $h : \tau$ or $M : \tau$. Even though a module h might be a module of implementation type Void.

A type environment Γ is a set of type assumptions representing a repository of typed modules. The details of the type rules, a more precise syntax, and reduction rules can be found in [6]. The type system presented in [6] guarantees that meta-computation computed using the reduction rules results in proper code with an implementation type. The program synthesis question is the type inhabitation problem: given a target type τ and a repository of modules Γ, decide whether there exists a composition of modules e that has type τ ($\exists e.\Gamma \vdash e : \tau$). We abbreviate the type inhabitation problem by $\Gamma \vdash ? : \tau$ and use an algorithm given in [6] to enumerate all compositions e (inhabitants).

6. SYNTHESIZING SEMANTIC COMPOSITIONS

We can now type modules and features according to the notions presented in the previous section and use type inhabitation for synthesizing type-safe compositions of derivatives and modules. We introduce two binary type constructors, *Code* and *Pt*. The constructor $Code(h, g)$ represents the disjoint union of modules h and g. The type of a derivative $\partial^n h/\partial F_1 \ldots F_n$ for a module h of implementation type τ_h is typed by the type constructor $Pt(\tau_h, F_1 \cap \ldots \cap F_n)$ with semantic types $F_1, \ldots, F_n$ representing n features. As in the works [1, 2], we can not avoid constructing all possible variations, in the worst-case exponential many, of modules and features since we can not restrict the space of compositions in general. But we follow the argumentation of Batory in [1] and observe that in real-world examples polynomially (often quadratically) many variations appear.

The operator $*$ of [2] is represented by the combinator $mult : \tau \to (\tau \to \sigma) \to \sigma$ taking a base module of type τ and an adaptor of type $\tau \to \sigma$ as arguments and returning their application as type σ. Operator $+$ is represented by the combinators:

$$\begin{aligned} AddConstr &: (\Box\tau) \to Code(\Box\tau) \\ Add2 &: (\Box\tau) \to (\Box\sigma) \to Code(\Box\tau, \Box\sigma) \\ Add &: (\Box\tau) \to Code(\Box\sigma, \Box\rho) \to \\ &\quad Code(\Box\tau, Code(\Box\sigma, \Box\rho)). \end{aligned}$$

Add allows to build list of pairs of independent (code) modules. We provide all combinators with implementations in $\lambda^{\Box}$-calculus as presented in [5] and [6]. For example, the application operator $*$ represented by the typed combinator $mult : \tau \to (\tau \to \sigma) \to \sigma$ and its implementation:

$$\begin{aligned} mult \triangleq \lambda x :\tau.\lambda y : \tau \to \sigma. \\ \text{letbox } X = x \text{ in letbox } Y = y \text{ in } Y(X) \end{aligned}$$

The operator $+$ is harder to type in simple types because the second argument is either a second derivative or a module. Using intersection types, we can easily intersect the types of both cases:

$$\begin{aligned} add : Pt(\tau, \phi_1) \to Pt(\tau, \phi_2) \to Pt(\tau_2, \phi_1 \cap \phi_2) \cap \\ Pt(\tau, \phi) \to \tau_h \cap \phi_h \to Pt(\tau, \phi \cap \phi_h) \end{aligned}$$

The composition of features $[H(f)] = h + \partial f/\partial H * f$ can now be typed as follows:

$$\begin{aligned} comp_{f,H,h} : \Box\tau_h \to \Box\tau_f \to Dt(\tau_f, \tau_H) \to \\ Code(\Box\tau_h, \Box\tau_f) \end{aligned}$$

We collect these type combinators for a feature model into a repository Γ for synthesis. The type inhabitation question for synthesizing two code modules h and g is:

$$\Gamma \vdash ? \; : \; Code(\Box\tau_h, \Box\tau_g).$$

It is guaranteed by the results of [6] that the returned code modules for h and g are well-typed modules of type τ_h and τ_g. We use the Combinatory Logic Synthesizer (CL)S [4] for synthesizing feature compositions that, when executed, generate well-typed software products fulfilling users' requirements. In case of over-specifying the synthesis goal, no compositions are returned.

We demonstrate the idea with a small example of a security module for hashing passwords with a password salt. The example is simplified but the generalization should be obvious, and the method is described in [6] in more detail. The module k has a code placeholder (**Salt**), where code of implementation type String is substituted, and computes a value of type Integer. Module k implements the computation of a cryptographic hash of a password as a STRING and returns the hash of the password as an Integer. The module is parameterized by a salt that modifies the string and can be used to prevent weak passwords. It has the following specification:

$$k : \Box\mathsf{String} \to \Box\mathsf{Integer}$$

$k \triangleq \lambda$salt : String.letbox **Salt**=salt in

```
box[
  Integer createHash(String message) {
  return Crypto.createHash(message+Salt);
  }]
```

We assume the method signature INTEGER CRYPTO.CREATEHASH(STRING). For our simplification, we assume that the salt module s only contains the string "SECRET" as a salt. Modules s has the following specification:

$$s : \Box\mathsf{String}$$

$$s \triangleq \text{box ["SECRET"]}$$

Collecting k and s in a repository $\Gamma = \{k, s\}$, we can use the type inhabitation question $\Gamma \vdash ? \; : \; \Box$Integer to receive the

applicative term $(k\,s)$ which is reduced using its implementations linked by $\triangleq$ to the term:

```
box[
  Integer createHash(String message) {
  return Crypto.createHash(message+"SECRET");
  }]
```

that is of type $\Box$Integer. When compiled and executed, the term returns an Integer representing a hash of password with a password salt. Although the example is very simple, the mechanism illustrated here allows in general arbitrary code substitutions and code transformations. Our Eclipse plugin LAUNCHPAD[1], a synthesis framework for feature-rich applications, for the FEATUREIDE supports more complex transformations.

7. CONCLUSION

We have a demonstrated how an algebraic formalism expressing feature compositions can be typed and transformed such that it is usable for automatic synthesis of feature- and module compositions with respect to feature- and module semantics. The composition is type-safe and results, after reduction, in compilable software product code.

Acknowledgment The authors would like to thank the anonymous reviewers for their valuable comments.

8. REFERENCES

[1] J. Liu, D. S. Batory, and S. Nedunuri, "Modeling Interactions in Feature Oriented Software Designs," in *FIW*, S. Reiff-Marganiec and M. Ryan, Eds. IOS Press, 2005, pp. 178–197.

[2] C. Prehofer, "Semantic reasoning about feature composition via multiple aspect-weavings," in *Proceedings of GPCE '06*. ACM Press, 2006, pp. 237–242.

[3] B. Düdder, M. Martens, J. Rehof, and P. Urzyczyn, "Bounded Combinatory Logic," in *Proceedings of CSL'12*, ser. LIPIcs, vol. 16. Schloss Dagstuhl, 2012, pp. 243–258.

[4] J. Bessai, A. Dudenhefner, B. Düdder, M. Martens, and J. Rehof, "Combinatory Logic Synthesizer," in *Proceedings of ISOLA'14*, ser. LNCS, vol. 8802. Springer, 2014, pp. 26–40.

[5] R. Davies and F. Pfenning, "A Modal Analysis of Staged Computation," *Journal of the ACM*, vol. 48, no. 3, pp. 555–604, 2001.

[6] B. Düdder, M. Martens, and J. Rehof, "Staged Composition Synthesis," in *Proceedings of ESOP'14*, ser. LNCS, vol. 8410. Springer, 2014, pp. 67–86.

[7] S. Apel, D. Batory, C. Kästner, and G. Saake, *Feature-Oriented Software Product Lines*. Springer, 2013.

[8] S. Apel and C. Kästner, "An Overview of Feature-Oriented Software Development," *Journal of Object Technology*, vol. 8, no. 5, pp. 49–84, 2009.

[9] D. Benavides, S. Segura, and A. Ruiz-Cortés, "Automated analysis of feature models 20 years later: A literature review," *Information Systems*, vol. 35, no. 6, pp. 615–636, 2010.

[1]Eclipse update-site available at http://combinators.org/launchpad/update-site

[10] M. Mannion, "Using First-Order Logic for Product Line Model Validation," in *Software Product Lines*. Springer, 2002, pp. 176–187.

[11] D. Batory, "Feature Models, Grammars, and Propositional Formulas," *Software Product Lines*, pp. 7–20, 2005.

[12] D. Benavides, P. Trinidad, and A. Ruiz-Cortés, "Automated Reasoning on Feature Models," in *Advanced Information Systems Engineering*. Springer, 2005, pp. 491–503.

[13] S. Thaker, D. Batory, D. Kitchin, and W. Cook, "Safe Composition of Product Lines," in *Proceedings of GPCE '07*. ACM, 2007, pp. 95–104. [Online]. Available: http://doi.acm.org/10.1145/1289971.1289989

[14] J. Rehof, "Towards Combinatory Logic Synthesis," in *BEAT'13, 1st International Workshop on Behavioural Types*, January 22 2013.

[15] J. Bessai, B. Düdder, A. Dudenhefner, T.-C. Chen, and U. de'Liguoro, "Typing Classes and Mixins with Intersection Types," in *Proceedings of ITRS'14*, J. Rehof, Ed., 2015.

[16] K. Czarnecki and A. Wasowski, "Feature Diagrams and Logics: There and Back Again," in *SPLC'07*. IEEE, 2007, pp. 23–34.

[17] N. Andersen, K. Czarnecki, S. She, and A. Wąsowski, "Efficient synthesis of feature models," in *Proceedings of SPLC'12*. ACM, 2012, pp. 106–115.

[18] C. Prehofer, "Feature-Oriented Programming: A Fresh Look at Objects," in *ECOOP*, 1997, pp. 419–443.

[19] D. Batory, P. Höfner, B. Möller, and A. Zelend, "Features, Modularity, and Variation Points," in *Proceedings of FOSD '13*. ACM, 2013, pp. 9–16.

[20] H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini, "A Filter Lambda Model and the Completeness of Type Assignment," *Journal of Symbolic Logic*, vol. 48, no. 4, pp. 931–940, 1983.

[21] B. Venneri, "Intersection types as logical formulae," *Journal of Logic and Computation*, vol. 4, no. 2, pp. 109–124, 1994.