

Language Support for Feature-Oriented Product Line Engineering

Wonseok Chae
Toyota Technological Institute at Chicago
wchae@tti-c.org

Matthias Blume
Google, Inc.
blume@google.com

ABSTRACT

Product line engineering is an emerging paradigm of developing a family of products. While product line analysis and design mainly focus on reasoning about commonality and variability of family members, product line implementation gives its attention to mechanisms of managing variability. In many cases, however, product line methods do not impose any specific synthesis mechanisms on product line implementation, so implementation details are left to developers. In our previous work, we adopted feature-oriented product line engineering to build a family of compilers and managed variations using the Standard ML module system. We demonstrated the applicability of this module system to product line implementation. Although we have benefited from the product line engineering paradigm, it mostly served us as a design paradigm to change the way we think about a set of closely related compilers, not to change the way we build them. The problem was that Standard ML did not fully realize this paradigm at the code level, which caused some difficulties when we were developing a set of compilers.

In this paper, we address such issues with a language-based solution. **MLPolyR** is our choice of an implementation language. It supports three different programming styles. First, its first-class cases facilitate composable extensions at the expression levels. Second, its module language provides extensible and parameterized modules, which make large-scale extensible programming possible. Third, its macro system simplifies specification and composition of feature related code. We will show how the combination of these language features work together to facilitate the product line engineering paradigm.

Categories and Subject Descriptors

D.2.13 [Software Engineering]: Reusable Software—*domain engineering*; D.3.2 [Programming Languages]: Language Classifications—*applicative (functional) languages, extensible languages, multiparadigm languages*

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.

FOSD'09, October 6, 2009, Denver, Colorado, USA.

Copyright 2009 ACM 978-1-60558-567-3/09/10 ...\$10.00.

General Terms

Design, Languages

Keywords

Feature-Oriented Programming, Product line engineering

1. INTRODUCTION

Product line engineering is a paradigm of developing a family of products [19, 22]. This emerging paradigm encourages developers to focus on developing a set of products rather than on developing one particular product. Therefore, we are expected to develop products from a common set of components (*core assets*) rather than from scratch. So far most efforts of working under this paradigm have focused on how to analyze a family of products and develop reusable assets. Features are commonly used to reason about commonality and variability in product lines. A set of features define a design space and the selection of a particular subset is the first step towards the synthesis of a design artifact [18].

In our previous work, we demonstrated that the product line engineering as a developing paradigm was a very effective way to build a family of compilers [12]. We showed engineering activities from product line analysis through product line architecture design to product line component design. Then, we presented how to build particular compilers from core assets resulting from such domain engineering activities. This approach especially helped us maintain source code by providing a systematic way to identify variations. This variability analysis provided us with the better chance of utilizing underline implementation technology. For example, the product line analysis enabled us to predict which parts would be mostly changeable, so we could parameterize over them via functorization provided by the Standard ML [5]. Then, the main program became a functor which took variations and produced a specific compiler. That is, we obtained a compiler generator. Product line engineering as a design paradigm was truly helpful here in that its product line analysis acts as a systematic way of identifying commonalities and variabilities. This made it possible to design and implement reusable and flexible software by supplying guidance on where relevant implementation techniques could be applied.

Although we have benefited from the product line engineering paradigm, it mostly served us as a design paradigm to change the way we think about a set of closely related compilers, not to change the way we build them. The problem was that our implementation technology did not fully

realize this paradigm at the code level. As a result, we experienced some difficulties:

- The relations among features and core assets (i.e., architecture and component models) were implicitly expressed only during the product line analysis as a form of documentation. Other product line model-based methods usually provide a way to express those relations explicitly by using CASE tools. In FORM, for example, those explicit relations make it possible to automatically generate product code from specifications [18].
- Our approach poorly supported systematic generation of family members.
- While we demonstrated our underline implementation technology (i.e., the Standard ML module system) was powerful enough to manage variations in the context of product lines, its type system sometimes imposed restrictions which caused code duplication between functions on data types.

This paper proposes a language support approach to address such limitations. Firstly, we will give a brief overview of the feature-oriented product line engineering (Section 2). As in our previous work, we will start with the feature modeling followed by product line asset design activities. Then, we will review the above issues in the context of product line implementation. Then, we present a language-based solution to better support product line implementation (Section 3). Our choice of an implementation language is **MLPolyR** [10]. It supports extensible programming in a compositional way and its module language provides parameterization mechanisms. Furthermore, its macro system provides feature-related code composition which is based on the feature selection. We will show how the combination of these language features work together to facilitate the product line engineering paradigm. Finally, we will review other feature-oriented implementation approaches (Section 4).

2. PRODUCT LINE ENGINEERING

Product line engineering highlights the development of products from core assets rather than from scratch. Therefore, this paradigm separates the process of building core assets from the process of building an individual product as shown in Figure 1. Product line asset development consists of analyzing a product line, designing reference architectures, and developing reusable components. In the product development process, a particular product is instantiated by selecting a proper architecture and adapting components. Among various product line approaches, we adopt FORM product line engineering for the following reasons:

- The method relies on a feature-based model which provides adequate means for reasoning about product lines [19].
- The method supports architecture design which plays an important role in bridging the gap between the concepts at the requirement level and their realization at the code level by deciding how variations are modularized by means of architectural components [25].

In this section, we will give an overview of overall engineering activities for developing a family of the simple arithmetic

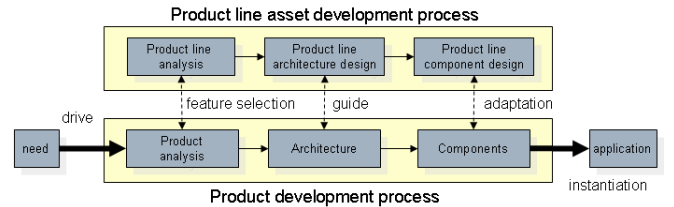


Figure 1: Development process (adopted from FORM [19]).

language interpreters. We believe that this simplified example can demonstrate the same limitation that our previous experience showed without dealing with unrelated technical details. Furthermore, this example is designed to precisely capture so-called *the expression problem* [29], which can even facilitate a comparative study of language support for product line implementation. The more detailed discussion of our choice of this example and the working draft of this comparative study can be found at our technical report [13].

2.1 Problem description

Let us consider a Simple Arithmetic Language (SAL) that contains terms such as numbers, variables, additions and a let-binding form. Suppose we start with two basic operations on terms: a function `eval` that realizes the evaluation semantics and a function `check` that realizes the static semantics. (In this case the static semantics just makes sure that all variables are in scope.)

In this setup, assume one wants to build an SAL interpreter `l`, which is the composition of the combinators `eval` and `check` where `o` means function composition:

$$l = \text{eval} \circ \text{check}$$

Sometimes, we want to replace an old implementation with a new one. For example, instead of the evaluation semantics `eval`, we can define a machine semantics (for example because we want to make control explicit) and implement its realization (`evalm`):

$$l_m = \text{eval}_m \circ \text{check}$$

Optionally, the combinator `opt` which performs some simple term rewriting (e.g., constant-folding, strength-reduction, etc.) may be inserted to build an optimized interpreter `lopt`:

$$l_{\text{opt}} = \text{eval}_m \circ \text{opt} \circ \text{check}$$

When the base language grows to support additional terms (e.g., a conditional term), `eval`, `opt` and `check` also evolve to constitute a new interpreter `l'opt`:

$$l'_{\text{opt}} = \text{eval}' \circ \text{opt}' \circ \text{check}'$$

Since all these interpreters have so much in common, we should be able to understand them as a *family* of interpreters. Therefore, it is natural to apply product line engineering for better support of their development.

2.2 Feature-oriented product line engineering

To analyze a set of interpreters as a family, we adopt feature-oriented product line engineering, which consists of following engineering activities:

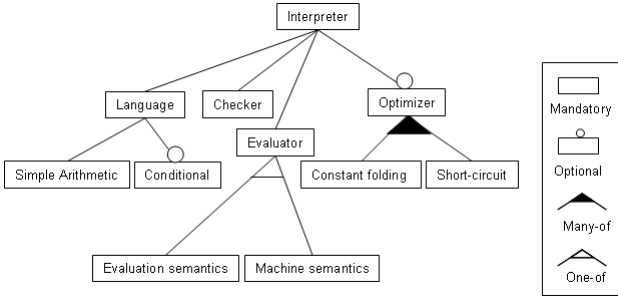


Figure 2: Feature model for the SAL interpreter. A closed triangle (many-of) represents multiple optional relationship and an open triangle (one-of) represents alternative relationship [15].

- *Product line analysis.* We perform commonality and variability analysis for the family of the SAL interpreters. We can easily consider features in the base interpreter as commonalities and exclusive features only in some extensions as variations. Based on this analysis, we identify two kinds of variations: architectural variation and component level variations. Then we determine which factors cause these variations. Such factors are represented as *features* in the feature model as illustrated in Figure 2.
- *Product line architecture design.* Architecture design involves identifying conceptual components and specifying their configuration. During this phase, we have to not only identify components but also define interfaces between components:

checker : term \rightarrow term
optimizer : term \rightarrow term
evaluator : term \rightarrow value

As usual, the arrow symbol \rightarrow is used to specify a function type. In our example, components act like pipes in a pipe-and-filter the architectural style, so all interface information is captured by the type.

By using the above conceptual components, we can specify the overall structure (i.e., *architecture*) of various interpreters:

interp = evaluator o checker
interpOpt = evaluator o optimizer o checker

- *Product line component design.* Next, we identify conceptual components which are constituents of a conceptual architecture. A conceptual component can have multiple implementations. For example, there are many versions of the *evaluator* component depending on the evaluation strategy:

eval : term \rightarrow value
eval_m : term \rightarrow value

At the same time, the language *term* can be extended to become *term'* which is an extension of *term* (for example to support conditionals):

eval' : term' \rightarrow value
eval'_m : term' \rightarrow value

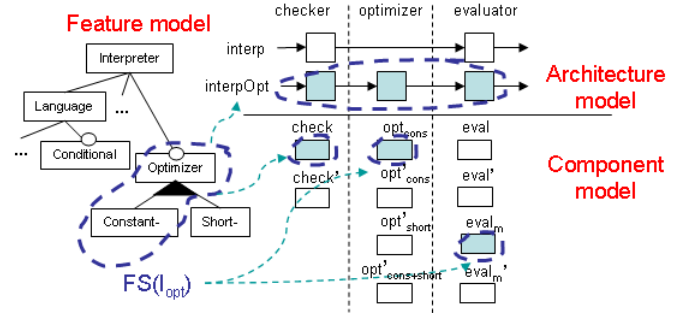


Figure 3: The overall product engineering process. Each selected feature gives advice on how to choose an architecture and how to instantiate required components. For example, the reference architecture *interpOpt* gets selected, guided by the presence of the *Optimizer* feature. The presence of the *Constant folding* feature guides us to choose the component *optimizer* with the implementation *opt_{cons}* [13].

Similarly, *check* and *check'* can be specified as follows:

check : term \rightarrow term
check' : term' \rightarrow term'

For the *optimizer* component, there are many possible variations due to inclusion or exclusion of various individual optimization steps (i.e., constant folding and short-circuiting) and due to the variations in the underlying term language (i.e., base and extended): *opt_{cont}*, *opt'_{cons}*, *opt'_{short}* or *opt'_{cons+short}*.

- *Product engineering.* Product engineering starts with analyzing the requirements provided by the user and finds a corresponding set of required features from the feature model. Assuming we are to build four kinds of interpreters, we have to have four different feature selections:

FS(I) = {Evaluation semantics}
FS(I_m) = {Machine semantics}
FS(I_{opt}) = {Machine semantics, Optimizer,
Constant folding}
FS(I'_{opt}) = {Conditional, Evaluation semantics, Optimizer,
Constant folding, Short – circuit}

Here, the function *FS* maps a feature product to its corresponding set of its required features. (For brevity only non-mandatory features are shown.) Selected feature sets give advice on the selection among both reference architectures and components. Figure 3 shows the overall product engineering process. The target product would be instantiated by assembling such selections.

2.3 Issues in product line implementation

During the product line asset development process, we obtain reference models which represent architectural and component level variations. Such variations should be realized at the code level. The first step is to refine conceptual architectures into concrete architectures which describe how to configure conceptual components. Then, product

line component implementation involves realization of conceptual components with the proper product feature delivery methods. Based on our experience of building a family of compilers [12], we see three issues that surfaced during product line implementation.

- *Product line architecture implementation.* Since there may be multiple reference architectures, it would be convenient to have mechanisms for abstracting architectural variations, capturing the inclusion or exclusion of certain components. Hence, any adequate implementation technique should be able to provide mechanisms for:

- declaration of required conceptual components (checker, optimizer and evaluator) and their interface
- specification of the base reference architecture `interp` and its optimized counterparts `interpOpt` by using such conceptual components.

Our experience showed that the Standard ML (SML) module language was powerful enough to describe both components and their interfaces. Each components in a reference architecture was mapped into a SML program unit (called *structure*) and interfaces among them as *signatures*. Furthermore, *functors* make it possible to implement the common part once and parameterize variations so that different products can be instantiated by assigning distinct values as parameters:

```

1 functor InterpFun(structure C : CHECKER
2                 structure E : EVALUATOR
3                 sharing C.T = E.T) : INTERP
4 where type term = E.T.term =
5 struct
6     type term = E.T.term
7     val interp = E.eval o C.check
8 end

```

Here, the functor `InterpFun` takes two components `checker` and `evaluator` and then returns their composition. However, all necessary sharing constraints between parameters must be declared explicitly. For example, in order to avoid type errors we had to specify that the types of `term` in both components should coincide.

- *Product line component implementation.* This phase involves realization of conceptual components. The main challenge of this phase is how to implement variations at the component level. Such variations could be in the form of either code extension or code substitution. For our running example, a pair of `check` and `check'` corresponds to code extension while a pair of `eval` and `evalm` corresponds to code substitution. While its parameterized module (i.e., functor) provides an efficient way to implement code substitution, the SML type system sometimes imposes restriction on code extension, which causes code duplication between functions on data types.
- *Product engineering.* Based on the product analysis, a feature product is instantiated by assembling product line core assets. For example, an extended interpreter I' can be instantiated by applying the functor

`InterpFun` to the modules `EChecker` (realizing `check'`) and `EBigStep` (realizing `eval'`) based on the feature selection. Each selected feature gives advice on how to choose an application architecture and how to instantiate required components. However, such instantiation was manually performed since the relations between features and core assets are implicitly expressed as a form of documentation.

3. LANGUAGE SUPPORT FOR PRODUCT LINE IMPLEMENTATION

In this section, we present product line implementation using `MLPolyR` as a language-based solution. `MLPolyR` is an ML-like language with row polymorphism, polymorphic record selection and polymorphic sums, functional record update and a Hindley-Milner-style type system with principal types [10, 11]. Among its many features, we focus on three aspects in this paper:

- *The extensible module language.* Similar to functors in the Standard ML module system, `MLPolyR` provides an parameterized mechanism called *template*. Furthermore, its modules are extensible and compiled separately.
- *Extensible programming with first-class cases.* With cases being first-class and extensible, one can use the usual mechanisms of functional abstraction in a style of programming that facilitates composable extensions. Note that these composable extensions are type-safe in a sense that well-typed programs do not go wrong [24].
- *The macro system.* Recent addition to `MLPolyR` supports code expansion at the level of a macro system. This mechanism makes it possible to write composition specification in terms of features, then feature selection will integrate the corresponding code easily.

In the remainder of this section, we will show how such mechanisms resolve each issue previously identified.

3.1 Product line architecture implementation

Each component in a reference architecture is mapped to an `MLPolyR` module. We first define types (or signatures) of the interested components based on the outcome of product line architecture design:

```

Checker    : {{ check : term → term, ... }}
Optimizer  : {{ opt   : term → term, ... }}
Evaluator  : {{ eval  : term → int,  ... }}

```

where `...` implies that there may be more parts in a component, but they are not our concerns. In practice, we do not have to write such interface explicitly since the type checker infers the principal types. Then, by using these conceptual modules (`Checker`, `Optimizer` and `Evaluator`), we can define two reference architectures:

```

1 module Interp = {{
2     val interp = fn e => Evaluator.eval
3                   (Checker.check e)
4 }}
5
6 module InterpOpt = {{
7     val interp = fn e => Evaluator.eval
8                   (Optimizer.opt
9                    (Checker.check e))
10 }}

```

Alternatively, like functors in SML, we can use a parameterized module called a *template* which takes concrete modules as arguments and instantiates a composite module:

```

1 template InterpFun (C, E) = {{
2   val interp = fn e => E.eval (C.check e)
3 }}
4
5 template InterpOptFun (C, O, E) = {{
6   val interp = fn e => E.eval
7     (O.opt
8      (C.check e))
9 }}

```

where C , O and E implicitly imply **Checker**, **Optimizer** and **Evaluator** respectively and their signatures are captured as constraints by the type checker. For example, the type checker infers the constraint that the module C should have a component named `check` which has a type of $\alpha \rightarrow \beta$ and β should be either an argument type of the module E (Line 1) or that of O (Line 5).

The second approach with templates supports more code reuse because a reference architecture becomes polymorphic because it is parameterized over its components including their types. As long as components satisfy constraints that the type checker computes, any components can be plugged into a reference architecture. For example, for the argument C , either the base module `Check` and its extension `EChecker` can be applied to the template `InterpFun`.

3.2 Product line component implementation

Modules in **MLPolyR** realize components. In order to manage component-level variations, we have to deal with both code extension and code substitution. For example, we will see multiple implementations of the component **Evaluator**:

```

BigStep   : {{ eval : term → int, ... }}
Machine   : {{ eval : term → int, ... }}
EBigStep  : {{ eval : term' → int, ... }}
EMachine  : {{ eval : term' → int, ... }}

```

where `term` represents a type of the base constructors and `term'` that of the extension. `BigStep` and `EBigStep` implement the evaluation semantics and its extension while `Machine` and `EMachine` implement the machine semantics and its extension. Note that a pair of `BigStep` and `EBigStep` (and also a pair of `Machine` and `EMachine`) corresponds to code extension while a pair of `BigStep` and `Machine` corresponds to code substitution.

Code extension is supported by first-class extensible cases. Figure 4 shows how such extensions are made. First, we define cases (Line 4-10) separately from a `match` expression (Line 14) where cases will be consumed. Then, we wrap such cases in functions by abstracting over their free variables (i.e., `eval` and `env`) (Line 3). One of these variables is `eval`—the whole evaluator itself. Its inclusion in the argument list achieves open recursion, which is essential to extensibility. With this setup, it becomes easy to add a new case (i.e., `IF0`). In an extension, only a new case is handled (Line 22-24) and the default explicitly refers to the original set of other cases represented by `BigStep.bases` (Line 25). Then, `EBigStep.bases` can handle five cases including `IF0`. We can obtain a new evaluator `EBigStep.eval` by closing the recursion through applying `bases` to evaluator itself (Line 29). Note that a helper function `run` is actually applied instead of `eval` in order to pass an initial environment in Line 30.

```

1 (* module for the evaluation semantics *)
2 module BigStep = {{
3   fun bases (eval, env) =
4     cases 'VAR x => env x
5         | 'NUM n => n
6         | 'PLUS (e1, e2) =>
7           eval (env, e1) + eval (env, e2)
8         | 'LET (x, e1, e2) =>
9           eval (Env.bind
10                (eval (env, e1), x, env), e2)
11
12   fun eval e =
13     let fun run (env, e) =
14         match e with bases (run, env)
15     in run (Env.empty, e)
16     end
17 }}
18
19 (* module for the extended evaluation semantics *)
20 module EBigStep = {{
21   fun bases (eval, env) =
22     cases 'IF0 (e1, e2, e3) =>
23       if eval (env, e1) == 0 then
24         eval (env, e2) else eval (env, e3)
25     default: BigStep.bases (eval, env)
26
27   fun eval e =
28     let fun run (env, e) =
29         match e with bases (run, env)
30     in run (Env.empty, e)
31     end
32 }}

```

Figure 4: The module `BigStep` realizes the evaluation semantics (`eval`) and the module `EBigStep` realizes the extended evaluation semantics (`eval'`) by defining only a new “conditional” case `IF0`.

Code substitution as another form of variation at the component level does not cause any trouble. For example, the module `Machine` implements the machine semantics (i.e., `evalm`). In our example two different implementations (`BigStep` and `Machine`) provide interchangeable functionality, but neither is an extension of the other, so they are implemented independently.

3.3 Product engineering

Our example asks us instantiate four interpreters (I , I_m , I_{opt} and I'_{opt}) differentiated by the feature selection. As Figure 3 demonstrates, each will be instantiated by selecting a proper architecture (either `InterpFun` and `InterOptFun`) and choosing its components (either `BigStep` or `Machine`, etc) with implicit advice from the selected feature set. For example:

- When the feature set is $FS(I_m)$, the reference architecture `InterpFun` gets chosen. Then, two components `Machine` and `Checker` are selected because of the presence of `Machine semantics` feature. Therefore, we instantiate the interpreter I_m as follows:

```

module Im = InterpFun (Checker, Machine)

```

- When the feature set is $FS(I'_{opt})$, the reference architecture `InterpOptFun` is chosen. As far as the components are concerned, the presence of the `Conditional` and `Evaluation semantics` features guide us to choose the component `EBigStep`. Similarly, the presence of

the **Optimizer**, **Conditional**, **Constant folding** and **Short-circuit** forces the use of component **ECSOptimizer** (realizing $\text{opt}'_{\text{cons+short}}$). Therefore, we instantiate the interpreter l'_{opt} as follows:

```
module  $l'_{\text{opt}}$  = InterpOptFun (EChecker,
                             ECSOptimizer,
                             EBigStep)
```

In this approach, we have to rely on implicit guidance on how to assemble core assets and then we manually perform a product instantiation since conventional languages cannot state the relations between a feature and its corresponding code segments in the program text. However, the **MLPolyR** macro system provides a way to explicitly specify the relations between features and core assets as a set of rules, so we can specify feature configuration in a separate file. One benefit of it is that the **MLPolyR** compiler can automate product engineering process, that is, it can automatically pick a right reference architecture and its associated components with respect to the given rules.

Figure 5 shows the expansion rules for a family of the SAL interpreters. It is used to map feature sets into feature-related abstractions (i.e., module names). For example, suppose the following macro term in the program text:

```
1 module Ix = @Interp
```

The macro term **@Interp** gets expanded recursively depending on the feature selection at the parse time. Let us assume that the selected feature set equals to $\text{FS}(l'_{\text{opt}})$. A reference architecture **InterpOptFun** will get selected (Line 4 in Figure 5) because of the presence of the **Optimizer** feature. Then, its conceptual component arguments **Checker**, **Optimizer**, **Evaluator** will be recursively instantiated into the proper concrete components. **Checker** will be expanded into a component **ECheck** due to the presence of **Conditional** (Line 10). **Optimizer** will be expanded into **ECSOptimizer** due to **Optimizer**, **Conditional**, **Constant folding** and **Short-circuit** (Line 15). **Evaluator** will be expanded into **EBigStep** due to the **Conditional** and **Evaluation semantics** features (Line 18). As a result, the term **@Interp** expands to the following expressions, which equals to the declaration of the module l'_{opt} that we had to manually write down:

```
1 InterpOptFun (EChecker, ECSOptimizer, EBigStep)
```

In summary, the **MLPolyR** compiler can expand macro terms with respect to rules, so the feature composition can be done automatically once we write expansion rules and provide a valid feature set.

4. DISCUSSION AND RELATED WORK

Our approach adopts FORM which aims to provide an unified feature-oriented development methodology [18, 19]. This method describes a mapping between features and core assets (i.e., design and implementation artifacts). However, it does not impose any specific synthesis mechanisms on its implementation, so implementation details are left to developers. Its early papers only illustrated the usage of the object-oriented programming annotated by the FORM macro language [18, 17] but recently, aspect-oriented programming has become popular as a way of implementing features in a compositional way [21, 14]. Our prior work on building a family of compilers introduced another choice

of implementation mechanism, i.e., functorization [12]. The contribution of this paper lies that we identify some difficulties of our previous approach and propose a direct language support for each phases in the product line implementation process:

- *Product line architecture implementation.* Similar to functors in SML, **MLPolyR** provides a parameterized module which has been demonstrated to be powerful to manage variations. Unlike SML, however, we do not have to write signatures explicitly since the **MLPolyR** type checker infers the principal types, which is expected to lessen the burden of product line implementors.
- *Product line component implementation.* The main challenge of this phase is to provide extensibility mechanism. Here, **MLPolyR** supports first-class cases that facilitate composable extensions.
- *Product engineering.* The **MLPolyR** macro system simplifies feature selection-based composition, so product instantiation can be automated once we supply a valid feature set.

While we aim to map between features in feature modeling and language constructs (realizing features), there have been attempts to support the concept of features more explicitly at code level. AHEAD, FeatureC++, CaesarJ and FeatureHouse are such feature-oriented programming languages that provide better abstraction and modularization mechanisms for features in various ways [8, 4, 6, 2]. Most these existing mechanisms fall into one of three categories:

- *The annotative approach.* This approach implements features using some form of annotations. Typically, preprocessors, e.g., macro systems, have been used in many literature examples. For example, the macro language in FORM determines inclusion or exclusion of some code segments based on the feature selection [18, 17].
- *The compositional approach.* In this approach, features are implemented as distinct units which are then integrated to become a product. Aspect-oriented or mixin-layered extension are such examples [8, 7].
- *The parameterization approach.* The idea of parameterized programming is to implement the common part once and parameterize variations so that different products can be instantiated by assigning distinct values as parameters. Higher-order modules, also known as *functors*—e.g., in SML are a typical example [5]. The SML module system has been demonstrated to be powerful enough to manage variations in the context of product lines [12].

Note that there have been hybrid attempts combining multiple different approaches [3, 20] and the **MLPolyR** language supports all three different programming styles. Similarly, FeatureC++ also supports all of them by combining aspect-oriented programming style, generic programming style (which enables parameterization over refinements) and an annotative approach [4]. However, its annotations are line-based in the style of **#ifdef** directives. Therefore,

```

1  (* Architecture Model *)
2
3  Interp ::= InterpFun    (Checker, Evaluator)    {}
4          | InterpOptFun (Checker, Optimizer, Evaluator) {Optimizer}
5
6
7  (* Component Model *)
8
9  Checker ::= Check      ()    {}
10           | ECheck     ()    {Conditional}
11
12 Optimizer ::= COptimizer ()    {Optimizer, Constant folding}
13           | ECOptimizer ()    {Optimizer, Conditional, Constant folding}
14           | ESOptimizer ()    {Optimizer, Conditional, Short-Circuit}
15           | ECSOptimizer ()    {Optimizer, Conditional, Constant folding, Short-Circuit}
16
17 Evaluator ::= BigStep   ()    {Evaluation semantics}
18           | EBigStep   ()    {Evaluation semantics, Conditional}
19           | Machine    ()    {Machine semantics}
20           | EMachine   ()    {Machine semantics, Conditional}

```

Figure 5: Expansion rules for the SAL product line. ::= denotes a “is-one-of” relation. For example, a macro term @Interp can be implemented by either a template InterpFun or InterpOptFun (Line 3-4). A template can have module arguments in the following (...). A module argument can recursively have its own expansion rules. Each rule defines the corresponding features in the list bracketed by {...}.

their feature specific code segments (if annotated) are scattered across multiple classes, so code easily becomes complicated. Saleh and Gomaa proposes the feature description language to overcome such problem [27]. Its syntax looks similar to the C/C++ preprocessor but it supports separation of concerns by modularizing feature specific code in a separate file. So does our macro system in that expansion rules are maintains in a separate file in order to prevent the growth of complexity.

Sunkle et al. proposed features as first-class entities [28] but our approach focuses on not features but relations between features and assets so that a user’s feature selection can pull the trigger at assembling, adapting and integrating core assets. Similarly, the AHEAD tools suite and FeatureHouse provide application generators but they treat a feature as a program increment or a program delta at the code level [23, 2]. These tools assume that there is a one-to-one mapping between conceptual *features* in feature modeling and concrete *features* in code. Therefore, in their approaches, the concepts at the requirement level and their realization at the code level should be similar while our approach studies features mainly in the requirement level and they are incrementally realized as assets during design and implementation phase. Therefore, the connection between mechanisms and each phase of the development process is of importance in our approach. This paper can be considered as an effort to make such connection explicit by providing relevant language supports.

The interpreter family example in this paper was originally designed to capture the so-called expression problem which describes the difficulty of two dimensional extensibility [13]. Variants of this problem have been popularly used to demonstrate the expressive power of feature-oriented programming. For example, as a variant, Lopez-Herrejon et al. defined the *Expression Product-Line (EPL)* to perform comparative study on feature modulization [23]. The aspect of generating different product-lines in the context of the expression problem has also been studied in the work

on Origami [9]. In an Origami matrix, orthogonal features are described as either rows or columns and they are synthesized through a series of matrix transformation to generate a product-line. Since Origami is not implementation-specific, we believe that their matrix transformation (i.e., folding a matrix) can be implemented using our language features in a way that rows and columns are represented by our extensible module language and extensible first-class cases and then our macro system performs a matrix transformation by assembling them.

Recently, Apel et al. discussed feature (de)composition in the context of functional programming [1]. However, their interest lies in the problem of cross cutting in functional programming while we are interested in applying functional programming techniques (e.g., higher-order modules and type-safe extensible cases) into feature composition.

5. CONCLUSIONS AND OUTLOOK

Our work shows that advanced programming language technology such as extensible cases and parameterized modules are helpful when we express and implement variations identified by product line analysis. Furthermore, our macro system simplifies feature selection-based composition. Although each of these mechanisms alone may make feature-oriented development more convenient, building a family of products becomes easier when all mechanisms are available.

We are continuing this work in several ways. First, we plan to integrate a feature modeling tool with our work. Since our expansion rules do not support any specification of feature relationships (i.e., mutually exclusive or required relations), the **MLPolyR** compiler cannot detect any invalid feature sets. We leave such validation to feature modeling tools which provide various diagnoses on feature models. Our goal is to let a front-end modeling tool generate valid expansion rules. Then, application engineering would only require feature selection.

We will also continue to improve architectural expressiveness of **MLPolyR** to enable it to describe various view-

points. For example, FORM supports three viewpoints (i.e., subsystem, process and module) [18]. So far only modules have been the focus in our architecture models. Note that our programming styles rely on the fact that there are two kinds of variations: architectural variations and component-level variations. We conjecture that different architecture styles (i.e., layered or blackboard style [16]) may require different programming models. In this line of research, we are investigating the usage of more expressive architecture description languages such as ArchJava which provides more explicit notations for dynamic configuration of product lines [26].

6. REFERENCES

- [1] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer. Feature (de)composition in functional programming. In *Proceedings of the 8th International Conference on Software Composition (SC)*, pages 9–26, July 2009.
- [2] S. Apel, C. Kästner, and C. Lengauer. FeatureHouse: Language-independent, automated software composition. In *Proceedings of the 31th International Conference on Software Engineering*, 2009.
- [3] S. Apel, M. Kuhlemann, and T. Leich. Generic Feature Modules: Two-Stage Program Customization. In *Proceedings of International Conference on Software and Data Technologies*, pages 127–132, Sept. 2006.
- [4] S. Apel, T. Leich, M. RosenmÄijller, and G. Saake. Featurec++: On the symbiosis of feature-oriented and aspect-oriented programming. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, pages 125–140, 2005.
- [5] A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In *Proceedings of the third International Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13, New York, Aug. 1991.
- [6] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. *An Overview of CaesarJ*. Lecture Notes in Computer Science : Transactions on Aspect-Oriented Software Development I, 2006.
- [7] AspectJ. <http://www.eclipse.org/aspectj/>, 2008.
- [8] D. Batory. Feature-oriented programming and the ahead tool suite. In *Proceedings of the International Conference on Software Engineering*, 2004.
- [9] D. Batory, R. E. Lopez-Herrejon, and J.-P. Martin. Generating product-lines of product-families. In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering*, page 81, 2002.
- [10] M. Blume, U. A. Acar, and W. Chae. Extensible programming with first-class cases. In *Proceedings of the International Conference of Functional Programming*, pages 239–250, 2006.
- [11] M. Blume, U. A. Acar, and W. Chae. Exception handlers as extensible cases. In *Proceedings of the ASIAN Symposium on Programming Languages and Systems*, 2008.
- [12] W. Chae and M. Blume. Building a family of compilers. In *Proceedings of the 12th International Software Product Line Conference*, 2008.
- [13] W. Chae and M. Blume. An evaluation framework for product line implementation. Technical Report TTIC-TR-2009, TTI at Chicago, 2009.
- [14] H. Cho, K. Lee, and K. C. Kang. Feature relation and dependency management: An aspect-oriented approach. In *Proceedings of Software Product Line Conference*, 2008.
- [15] K. Czarnecki and U. W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., 2000.
- [16] D. Garlan and M. Shaw. An introduction to software architecture. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1994.
- [17] K. C. Kang, M. Kim, J. Lee, and B. Kim. Feature-oriented re-engineering of legacy systems into product line assets - a case study. In *Proceedings of the Software Product Line Conference*, pages 45–56, 2005.
- [18] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. Form: A feature-oriented reuse method with domain-specific reference architectures. *Ann. Softw. Eng.*, 5:143–168, 1998.
- [19] K. C. Kang, J. Lee, and P. Donohoe. Feature-oriented product line engineering. *IEEE Softw.*, 19(4):58–65, 2002.
- [20] C. Kästner and S. Apel. Integrating compositional and annotative approaches for product line engineering. In *Proceedings of the GPCE Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering (McGPLE)*, pages 35–40, Oct. 2008.
- [21] K. Lee, K. C. Kang, M. Kim, and S. Park. Combining feature-oriented analysis and aspect-oriented programming for product line asset development. In *Proceedings of the 10th International on Software Product Line Conference*, pages 103–112, 2006.
- [22] K. Lee, K. C. Kang, and J. Lee. Concepts and guidelines of feature modeling for product line software engineering. In *Proceedings of the 7th International Conference on Software Reuse*, pages 62–77, 2002.
- [23] R. E. Lopez-herrejon, D. Batory, and W. Cook. Evaluating support for features in advanced modularization technologies. In *European Conference on Object-Oriented Programming*, 2005.
- [24] R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.
- [25] N. Noda and T. Kishi. Aspect-oriented modeling for variability management. In *Proceedings of the International Software Product Line Conference*, 2008.
- [26] S. Pavel, J. Noyé, and J.-C. Royer. Dynamic configuration of software product lines in archjava. In *Proceedings of Software Product Line Conference*, pages 90–109, 2004.
- [27] M. Saleh and H. Gomaa. Separation of concerns in software product line engineering. *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, 2005.
- [28] S. Sunkle, M. Rosenmüller, N. Siegmund, S. S. ur Rahman, G. Saake, and S. Apel. Features as first-class entities - toward a better representation of features. In *Proceedings of the GPCE Workshop on Modularization, Composition, and Generative Techniques for Product Line Engineering (McGPLE)*, pages 27–34, Oct 2008.
- [29] P. Wadler. The expression problem, Dec. 1998. Email to the Java Genericity mailing list.