ALGEBRA: A Programming Language for Developing Software Product Lines based on Object Algebras

November 11, 2013

1 Abstract

A Software Product Line (SPL) is a family of software products sharing a set of common features, where each product may introduce a degree of variability. State-of-the-art tools in SPL development, including AHEAD and FeatureHouse, use a meta-programming approach based on code generation. That is, there is some tool/(meta-)language that is used to describe the SPL and when a particular product is needed, the tool will generate code in a target language (such as Java or C++). Unfortunately, this meta-programming approach of developing SPLs has important drawbacks, including the lack of separate compilation and modular type-checking. The lack of separate compilation prevents a business model where companies would sell the binaries of the SPL modules to customers. For companies this is an attractive business model because it allows customers to build customized products, without giving these customers access to the source code (and to the company's intellectual property). The lack of modular type-checking makes the development process more complex and brittle. For example, in terms of error reporting, type-errors can only be reported after a concrete product has been generated. Furthermore these errors would not be reported back to the developer in terms of the code he has written, but rather in terms of the generated code. This is clearly unsatisfactory.

Our project aims at addressing the drawbacks of meta-programming approaches by creating ALGEBRA: a new programming language for developing SPLs. ALGEBRA does not take a meta-programming approach. Instead it takes a programming approach, where the mechanisms for the creation of SPLs are directly integrated in the semantics of the language. As a consequence programs written in ALGEBRA fully support modular type-checking and separate compilation. To achieve this, ALGEBRA will have new language constructs that provide built-in support for object algebras. Object algebras are a recent development in object-oriented programming that can be used to develop highly modular programs. The ALGEBRA programming language will be the first programming language to provide native support for object algebras, allowing: 1) convenient development of highly modular programs and SPLs; 2) efficient compilation of programs using object algebras. At the end of the project we expect that the state-of-the-art in SPL development will be made dramatically more efficient from a developer productivity perspective while being significantly more attractive from a business perspective.

2 Impact and Objectives

Long-term impact

It has been widely acknowledged that existing language/tool support for SPL development is unsatisfactory, due to lack of features such as separate compilation and modular type-checking. Our project is innovative and significant in that it addresses these important concerns by developing the necessary programming language technology. To the best of our knowledge, this is the first major project for a programming language that supports the development of SPLs, while at the same time, featuring modular type-checking and separate compilation. This is required to enable companies to develop SPL technology and commercialize the corresponding modules, without having to give the source code to customers. Such business model could be useful in the context of business applications. By packaging their business application as a SPL, each individual customer could easily customize the business application to their own needs. Furthermore, modular type-checking will be important for the adoption of SPL technology by developers, since it will make the development process easier and more robust. Namely with respect to error reporting, errors will be caught when compiling the SPL modules and will be reported back in terms of those modules. Our project provides an important step in that direction and it is likely to significantly improve the state-of-the-art in SPL development.

From a more general point of view, our project will also offer a new perspective and approach on how to deal with the important problem of crosscutting concerns in software. A crosscutting concern is code that logically represents some separate, modular functionality of a program, but cannot be implemented in a modular way and ends up scattered across an existing code base. The presence of crosscutting concerns in software is detrimental to software quality because code becomes harder to reason about, reuse and maintain. Crosscutting concerns are pervasive in SPLs and are the reason why SPLs cannot be easily implemented directly in existing programming languages. For the last 20 years, the Programming Language and Software Engineering communities have identified crosscutting concerns as being a major challenge, and have widely acknowledged that existing PL abstractions are insufficient to deal with such concerns. Several high-profile works — including "Aspect-Oriented Programming" (currently the most cited publication in Programming Languages according Microsoft Academic Search); and the 10-year award most influential paper award for ICSE 1999 "N degrees of Separation: Multi-Dimensional Separation of Concerns" — have proposed solutions to deal with the issue of crosscutting concerns. However, such solutions, much like existing SPL solutions, tend to use meta-programming based techniques, which offer little or no support for separate-compilation and modular type-checking. As such the outcome of our project can offer significant new insights on how to deal with crosscutting concerns while still supporting separate compilation and modular type-checking.

Finally, this research can also have a significant impact in both the implementation and teaching of programming language technology. From the implementation point of view, the technology developed in this project will allow obtaining reuse across similar languages or languages with different extensions. A concrete example would be a SPL capturing reusable components for compilers or program analyzers. There are many languages out there (for instance Java, C# or C++) which require a lot of similar infrastructure for such components. Ideally such infrastructure should be captured in a reusable form as a SPL, and then the particular variations specific to each language could be added later. From the teaching point of view, having language technology that allows modular development of compilers and program analyzers will provide ideal tool support for most existing courses on programming languages or compilers. The curriculum of such courses typically starts with a very simple language to which more complex features are added over time, until at the end a relatively realistic language is fully developed. Good tool support for SPL technology will allow the course material to be incrementally developed by the students and multiple implementations of variants of programming languages to co-exist and reuse the same code. We plan to use this technology in the local HKU courses on Principles of Programming Languages (CSIS0259) and Compiling Techniques (CSIS0235).

Objectives

- Native language support for SPL development using Object Algebras: Our project will study how to provide native language support for object algebras, including syntax, modular type-checking and separate compilation.
- Study of compilation schemes for ALGEBRA: We will study three different alternative compilation schemes for ALGEBRA, which will offer different trade-offs in terms of modularity, efficiency and simplicity of implementation.
- Development of case studies and evaluation of the approach: To illustrate the applicability and benefits of ALGEBRA, we will develop several SPL case studies and compare our results with other SPL tools.

3 Background of Research

Programming languages play a fundamental role in the development of reliable, maintainable and performing software. Since the dawn of computer programming, there has been a constant search for better programming language abstractions to structure programs. This has led to groundbreaking developments such as the introduction of *structured programming* [24, 35], which in turn has promoted the development of even higher-level abstractions including *objects* [31], *modules* [53] and *abstract datatypes* [49]. All of these developments have played a crucial role in enabling the development of the ever increasing more complex software that we see today.

3.1 Crosscutting concerns

The natural question to ask is: "what's next"? Researchers have been actively looking for the next higher-level abstractions which will enable even better structuring of programs. Over the last 20 years there have been several proposals which have raised considerable interest from both the research community and industry. Such proposals include Aspect-Oriented Programming (AOP) [47], Feature-Oriented Programming (FOP) [59, 41, 2], Subject-Oriented Programming (SOP) [42] or Multi-dimensional Separation of Concerns [65]. All such proposals agree that a problem with existing programming language abstractions is that they often give rise to *crosscutting concerns*, and each approach offers the promise to address that problem in its own way. A crosscutting concern is code that logically represents some separate, modular functionality of the program, but is scattered across an existing code base. There are different types of crosscutting concerns [45], including:

- **Homogeneous crosscutting concerns:** This type of crosscutting concerns, which has been popularized by Aspect-Oriented Programming, corresponds to repeated occurrences of similar pieces of code throughout a code base. Common examples include code for *logging*, *profiling*, or *security-related* code.
- **Heterogeneous crosscutting concerns:** This type of crosscutting concern differs from homogeneous concerns, in that *different* pieces of code are required at different locations. A typical example is the addition of a new method to an existing class hierarchy.

Figure 1 illustrates two concrete examples where the two types of crosscutting concerns arise. Both examples are based on a simple form of expressions that could be used, for instance, in an interpreter or compiler. The left-side of the figure illustrates a situation where the programmer intends to add a simple tracing mechanism for monitoring the methods being called. To implement this, the programmer would need to create many instances of similar code throughout the various methods (intended to be traced) in the class hierarchy. This situation is a typical example of a homogeneous crosscutting concern. The right-side of the figure illustrates another situation where the programmer intends to add an evaluation feature for expressions. This situation is different from tracing because, although code also needs to be added in multiple locations, this code tends to be quite different for each location. This is a good illustration of a heterogeneous crosscutting concern.

In general, the presence of crosscutting concerns in software is detrimental to software quality because code becomes harder to reason about, reuse and maintain. Consequently previous research has focused on providing mechanisms that allow modularizing such crosscutting concerns.

3.2 Heterogeneous Crosscutting Concerns and Software-Product Lines

The main objective of this proposal is to create programming language technology for developing *Software-Product Lines* (SPLs) [29, 58]. A SPL is a family of software systems sharing a set of common features. Each system may introduce a degree of variability. The development of SPLs typically gives rise to *heterogeneous* crosscutting concerns [45]. Currently there are two alternatives for dealing with heterogeneous crosscutting concerns: *meta-programming-based* and *programming-based* approaches.

Meta-Programming Based Approaches For the purposes of this proposal what we mean by meta-programming [63] is the use of tools or languages that generate or manipulate code written in some other language. For example the AspectJ [46] language, which is an AOP extension of Java, can be compiled by generating plain Java code or by modifying existing Java bytecode. In other words AspectJ is a meta-language that generates (or manipulates) Java (byte)code.

The appeal of meta-programming techniques lies in their expressive power. Through meta-programming it is possible to do non-trivial manipulations of the syntax of some target language, from adding new code to a method, to creating new methods or classes. Most major proposals for new modularization mechanisms targeting crosscutting concerns — including AOP [47], FOP [59, 41, 2], SOP [42] or Multi-dimensional Separation of Concerns [65] — take a meta-programming approach. AOP and FOP are particularly relevant to us, since AOP is perhaps the most well-known modularization technique and FOP is targeted at the development of SPLs.

Aspect-oriented (meta-)languages like AspectJ are particularly good at dealing with *homogeneous* crosscutting concerns. Homogeneous concerns such as tracing or logging are decoupled from a code base and captured in a single location, in a modular way, using a AOP language construct called an *aspect*. A tool like AspectJ then takes those aspects as well as an existing Java code base and "weaves" in the aspect code into appropriate locations in the code base. AOP languages like AspectJ can also deal, to some extent, with heterogeneous crosscutting concerns. However, several researchers have reported that the development of SPL features with AOP languages is unsatisfactory [50, 45]. Issues include the lack of support to group multiple cooperating aspects and classes belonging to a feature together, and the unsuitability of AOP-constructs to deal with certain heterogeneous crosscutting concerns.

Feature-oriented programming has been promoted as a paradigm for the development of SPLs [8, 5, 4]. Several mature FOP tools have been created with their own (meta)-language mechanisms to express features. Examples of such tools are GenVoca [9], AHEAD [8], FeatureC++ [5] and FeatureHouse [4]. Language mechanisms used in these tools include *collaborations* [68, 64], *Mixin layers* [64], and extensions like *aspectual mixin layers* [6]. Many of these tools and language mechanisms are normally implemented via *superimposition*. Superimposition is a simple meta-programming technique that merges two components with the same name. For example given two modules with a class named *Person*, superimposition would create another *Person* class resulting from the merged contents of the two original classes. Some recent work on FOP has focused on creating a general, language independent model of superimposition that can be applied to different target languages [7, 4].

While meta-programming approaches are very powerful and expressive, they have important drawbacks. Since these approaches are very syntactic in nature (they just manipulate the syntax of a target language), it is hard to get modularity benefits at a more semantic level. For example, both type-checking and separate compilation (which are very semantic properties) are usually not supported by tools that rely on meta-programming techniques such as superimposition.

Programming Based Approaches An alternative approach is to build-in constructs that deal with crosscutting concerns directly in a programming language. That is, instead of having two languages (a meta-language and a target language), only one language is needed to deal with crosscutting concerns. The challenge in this approach is how to integrate the modularity constructs in the semantics of the programming language, so that crosscutting concerns can be dealt with directly. In particular, it is necessary to provide the language with a static semantics (the type system) and a dynamic semantics (the run-time system). Both the static and dynamic semantics are expected to be modular (leading to modular type-checking and separate compilation). Unfortunately, this is a non-trivial problem, as witnessed by well-known programming language challenges such as *the expression problem* [71], or observations by many researchers on the inability of existing languages to deal with more than one dimension of modularity at once [42, 47, 65]. Nevertheless there have been some interesting proposals that we discuss next.

The SPL and FOP communities have acknowledged the need for more semantic properties, including type-checking, program analysis and testing of SPLs [2]. Ideally it should be possible to do all of these without generating all the members of a SPL. Thaker et al. [66] present an approach for type-checking AHEAD SPLs. This approach is not completely modular as some global checks are necessary. There have also been proposals [3, 61, 32] for FOP-related calculi, which build FOP modularity constructs directly into the semantics. However type-checking is still not completely modular, and the cost of type-checking grows proportionally to the number of possible products in the SPL. Other authors have also investigated how to do program analysis [22] and testing [48] of SPLs efficiently.

In the programming language community, there has also been some work related to modularity of crosscutting concerns, although not directly targeted at SPLs. Family polymorphism [37] is a proposal for a language mechanism that can group complete hierarchies of classes into a family. With family polymorphism when software evolves, classes in the same family evolve together. An example would be the simple hierarchy of expressions in Figure 1. If a new method is added to the interface of expressions, then all classes implementing that interface need to evolve together by adding a new method. *Virtual classes* [38, 55] are a concrete mechanism that implements family polymorphism. In a language with virtual classes, classes can have nested classes and, just like methods, such nested classes can be overriden (that is they are virtual). Modular type checking of virtual classes is possible, but very subtle. Only relatively recently a type-sound calculus supporting modular type-checking of virtual classes has been proposed [38]. There are also other mechanisms, such as *virtual types* [27], which are closely related to virtual classes and for which modular type-checking is less involved. Virtual types can also model family polymorphism and the Scala language supports a variant of virtual types. Unfortunately, modularizing crosscutting concerns with virtual types tends to be quite heavyweight, requiring considerable boilerplate code for composition of features and complex types for the features [57]. Another alternative way to encode family polymorphism, and modularize crosscutting concerns, is to use *F-bounded polymorphism* [28] as present in languages like Java or C# [67, 60]. Much like virtual types, F-bounded encodings of family polymorphism tend to be heavy and the types are particularly difficult to grasp for most programmers.

In summary, while the programming based approaches discussed here are promising, they are also significantly more challenging than meta-programming approaches to implement. Some calculi do support modular type-checking [38], but the algorithms are quite subtle and complex. Some other approaches support type-checking, but not in a completely modular way [3, 61, 32]. Furthermore from the implementation point-of-view having a calculus still leaves the question of how languages based on that calculus can be effectively implemented. Building a virtual machine or run-time system is possible, but the engineering effort in making these robust and efficient enough to compete with mature virtual machines like the JVM is considerable. Alternatively compilation strategies that target existing virtual machines need to be developed.

3.3 Work Done by the investigator

Dr. B. C. d. S. Oliveira has published several full papers in leading venues of Programming Languages (and closely related areas). These venues include *PLDI* [18], *POPL* [33], *ECOOP* [10, 11, 13], *OOPSLA* [21, 17], *ICFP* [62, 12, 34], *JFP* [40, 15, 20], *ICSE* [25], *FSE* [26] and *AOSD* [19]. Most of his papers are related to modularity (which is the main topic of this proposal) and he has received 2 best paper awards, including the best paper for ECOOP 2012.

The main question addressed in his previous work is to which extent can we use *existing* programming languages to deal with crosscutting concerns? The rationale is that by using existing programming languages we get separate compilation and modular type-checking for free. In the ECOOP 2012 paper [11] and a follow up paper [13], Oliveira and colleagues have shown that it is possible to modularize crosscutting concerns using existing mainstream languages (such as Java, C# or Scala) and using a relatively conventional OO type system with generics. This was a significant advance, because *all previous approaches*, could only achieve modularity with new programming languages or advanced programming language features (such as F-bounded quantification, wildcards and variance annotations). The key to this achievement was the use of a new abstraction, called *object algebras*, which allows developing programs that are more modular than conventional object-oriented programs using a *design pattern* [39]. Object algebras are an evolution of a line of work [14, 16, 21, 10], pursued by the author and collaborators, which exploits Church encodings of datatypes [23] to overcome modularity issues in programming languages. The author and collaborators have also shown that such Church-encoding inspired techniques scale up to solve non-trivial problems such as the modularization of programming language meta-theory [33, 34].

Of course, by restricting ourselves to existing PLs, we can only program with object algebras using a design-pattern-based encoding. Thus, while programming with object algebras in existing languages is possible, it is not very convenient to use since considerable amounts of boilerplate code are needed. Moreover the encodings have limited expressiveness and introduce abstraction overhead which affects performance.

4 Research Plan and Methodology

We propose the ALGEBRA programming language to develop Software Product Lines. ALGEBRA follows a programming language approach to SPLs, and has full support for modular type-checking and separate compilation. However, unlike other programming-based approaches to modularity, ALGEBRA will support simple, convenient and expressive language abstractions to model the reusable components of SPLs. ALGEBRA's approach to modularity will be based on object algebras. The main advantage over our previous work will be first-class support for object algebras: there is no need for heavy encodings using a design pattern. This will allow a much more direct and convenient way to express programs with object algebras; and support significantly more efficient compilation methods.

4.1 Motivating Example: The Expression Product Line

We use the *Expression Product Line* (EPL) [52] to illustrate ALGEBRA and its use to develop SPLs. The EPL is a canonical product line and a standard benchmark in SPLs technology. This product line illustrates *some* of the basic challenges in developing heterogeneous crosscutting concerns modularly. The goal is to represent data types for expressions of the form:

```
Exp  :: = Lit | Add | Neg
Lit  :: = <non-negative integers>
Add  :: = Exp "+" Exp
Neg  :: = "-" Exp
```

There are three types of expressions: literals (non-negative integers); addition of expressions; and negation of expressions. Furthermore, there are two possible operations: *evaluation* of expressions (*Eval*), and textual *printing* of expressions (*Print*). The types of expressions $\{Lit, Add, Neg\}$ and the operations $\{Eval, Print\}$ constitute two different feature sets. The challenge is to allow any possible combination of features while at the same time reusing

the common parts of the code. We assume that the *Print* and *Lit* features must always be present. As such, there are 8 possible combinations, which are illustrated in Figure 2.

Figure 5 shows a complete solution to the EPL using ALGEBRA. The details of the language will be discussed in Section 4.2, but the important thing to note now is that the code modularizes the different features of the EPL in a *reusable* and *convenient* way. The first two declarations (the interfaces *IEval* and *IPrint*) are just conventional object-oriented interfaces declaring methods *eval* and *print*. These declarations correspond to the interfaces of the operation features *Eval* and *Print*. ALGEBRA introduces a number of language constructs such as **algebra interface**, **algebra** and **data** declarations. These declarations allow convenient syntactic sugar for the definition of object algebra interfaces, object algebras (and corresponding constructor implementation) and objects constructed from object algebras. The algebra interfaces *ExpAlg*, *AddAlg* and *NegAlg* provide the interfaces for the three types of expression features ($\{Lit, Add, Neg\}$). New syntax, such as $print@(Lit\,x)$, provides a convenient way to implement the feature methods in the algebras. This syntax is inspired by pattern matching from functional programming languages like Haskell.

All the definitions can be type-checked and compiled modularly and particular products can be built from these definitions. For example, in the *main* function two objects that correspond to two different products are used. The first object (o_1) is based on a product that supports the *Print*,*Lit* and *Add* features. The second object (o_2) is based on a product that supports the *Eval*,*Print*,*Lit*,*Add* and *Neg* features. Syntax such as $exp_1\langle Print\rangle$ allows objects of a certain datatype (in this case *Exp*) to implement a compatible object algebra (in this case *Print*). This provides those objects with the methods defined by the object algebra. Moreover it is possible for objects to implement multiple algebras. This is illustrated in the expression $Exp\langle NegPrint, Eval\rangle$, which uses both the *NegPrint* and *Eval* algebras. The types of objects built using multiple algebras have *intersection types* [30] such as *IEval* **with** *IPrint*. Objects for all other 6 possible combinations of the EPL can be created similarly to o_1 and o_2.

4.2 Project Plan

This project will run for three years. We divide the project into three sub-tasks. The project will require three members: the Principal Investigator (P); and two graduate students (Student A and Student B). Figure 4 provides an overview of the project schedule. The remainder of this section provides the details of all tasks in the project.

4.2.1 Task 1: Language Design and Formalization

The first task of this project is to study the language design and formalization of ALGEBRA to enable native language support for object algebras. This will include formalizing and designing all aspects of the syntax, static semantics, dynamic semantics, investigating useful language extensions like generics, and also proving the relevant theorems to establish type-safety. The careful formalization of all meta-theory aspects of the syntax, semantics, and extensions to ALGEBRA is a major undertaking that will be the basis of one graduate student's work (Student A).

As preliminary work for this task, we developed a basic source syntax for ALGEBRA that is sufficient to capture the code in Figure 5, and sketched the form of the relevant theorems to establish type-safety. We believe that this is sufficient to illustrate the advantages of first-class language support for object algebras, and to illustrate the feasibility of this task.

Task 1.1: Syntax and Semantics Formally, we can capture the key constructs of a simplified version of ALGEBRA (enough to encode the example in Figure 5) by the following grammar:

Declarations

Program	::=	$\overline{Decl}$	Program
Decl	::=	$I \mid AI \mid A \mid D \mid \ldots$	Declaration
I	::=	**interface** N_I **extends** $\overline{N_I}$ **where** $\overline{ms}$	Interface
AI	::=	**algebra interface** N_{AI} **extends** $\overline{N_{AI}}$ **where** $\overline{\textbf{sort}\ S\ \overline{cs}}$	Algebra Interface
A	::=	**algebra** N_A **extends** $\overline{N_A}$ **implements** $\overline{N_{AI}}$ **where** $\overline{\textbf{sort}\ S = N_I}$ $\overline{m@(C\ \bar{x}) = E}$	Algebra
D	::=	**data** N_D **from** $N_{AI}.S$	Datatype
...			

Types

T	::=	T **with** T	Intersection Type
	$\mid$	...	

Expressions

E	::=	$E\langle\overline{N_A}\rangle$	Algebra instantiation
	$\mid$	...	

This grammar contains various types of declarations that we expect to have in ALGEBRA, and provides a simple model of types and expressions. The notation $\bar{x}$ means a sequence of x's. For space reasons, we show only the important and novel language constructs here. In particular algebra interfaces (AI), algebras (A) and datatypes (D) in combination with a special expression for the composition of algebras ($E\langle\overline{N_A}\rangle$), and intersection types (T **with** T),

provide first-class support for the definition and composition of object algebras. Algebra interfaces resemble typical OO interfaces (such as the ones found in Java). Like OO interfaces they can extend multiple other (algebra) interfaces. However, unlike OO interfaces, algebra interfaces declare sorts ($\overline{\textbf{sort}\ S}$) and constructors ($\overline{cs}$). Sorts, such as *Exp* in Figure 5, are abstract types that are used by the constructors. In particular, the return type of a constructor must always be a sort type. Algebras implement algebra interfaces by instantiating the sorts to some interface type ($\overline{\textbf{sort}\ S = N_I}$) and providing the method implementations for those interfaces for each constructor in the algebra ($\overline{m@(C\ \overline{x}) = E}$). Data declarations, allow declaring a top-level type N_D, that can be used to build data of a particular sort (S) in an object algebra interface (N_{AI}). Algebra instantiation ($E\langle\overline{N_A}\rangle$) takes an expression whose type is a datatype N_D and instantiates the resulting object with a set of algebras ($\overline{N_A}$). Finally, intersection types ($T\ \textbf{with}\ T$) are used to type the instantiated objects that implement multiple algebras.

Semantics An important component of ALGEBRA will be its modular type-system (static semantics). The modular type-system will enable a dynamic semantics via a type-directed translation. This style of dynamic semantics in terms of a type-directed translation is also used by well-known language mechanisms, including *type classes* [70] or *implicits* [17, 18]. Furthermore the type-directed translation will enable a first compilation scheme for ALGEBRA, which is discussed in more detail in Section 4.2.2.

A key design concern for the type system is to allow modular components to have simple types, which can be understood by mainstream developers. ALGEBRA's type system will be based on a relatively conventional type system with intersection types [30]. To design the type system, we can build on the knowledge that we have accumulated in our previous work on encoding object algebras (and corresponding composition operators) into languages like Java or Scala [10, 11, 13]. Our experience suggests that the type system will be quite similar to type systems like that of Featherweight Java [44], except for the inclusion of intersection types. As illustrated in our previous work [13], intersection types ($T\ \textbf{with}\ T$) are important to express the types of objects constructed from composed algebras, as in the object o_2 in Figure 5.

Task 1.2: (Mechanical) Formalization and Proofs We will build on our previous experience with programming language meta-theory formalization [33, 34], and use the Coq theorem prover to formally verify all aspects of ALGEBRA's semantics and meta-theory. To formalize the type-directed translation more precisely (for a subset of ALGEBRA) we can use Featherweight Generic Java (FGJ) [44] as a target language. This will allow us to prove important theorems. Namely, we will prove that type-directed translation is type-preserving:

Theorem 4.1 (Type-preserving translation) *Let E be an ALGEBRA expression, T be an ALGEBRA type and e be a FGJ expression. If $\cdot \vdash_{ALGEBRA} E : T \rightsquigarrow e$, then $\cdot \vdash_{FGJ} e : |T|$.*

This theorem states that well-typed expressions in ALGEBRA translate into well-typed FGJ expressions e with type $|T|$, where $|T|$ is the translation of ALGEBRA's type T into a FGJ type. The dynamic semantics of an ALGEBRA expression ($eval(E)$) is the composition of the type-directed translation and FGJ dynamic semantics.

$$eval(E) = V \qquad where \cdot \vdash_{ALGEBRA} E : T \rightsquigarrow e \ and \ e \rightarrow^* V$$

with $\rightarrow^*$ being the reflexive, transitive closure of FGJ standard single-step call-by-value reduction relation. This allows stating the conventional type safety theorem for ALGEBRA, which should easily follow from type-preservation:

Theorem 4.2 (Type Safety) *If $\cdot \vdash_{ALGEBRA} E : T$, then $eval(E) = V$ for some FGJ value V.*

Task 1.3: Language Extensions To ensure that ALGEBRA provides language support for realistic software development, it is also necessary to investigate the interaction of the new language constructs with other, more conventional programming language features. From our experience encoding object algebras in languages like Java, we do not expect challenges in adding conventional language constructs for imperative programming and object-oriented programming (such as classes). However, we do expect some challenges integrating generics in ALGEBRA. This is because sorts, much like generics, also provide a form of type-parametrization. Thus abstracting over object algebra types with sorts seems to intrinsically need higher-order type parametrization. This is problematic for our type-directed translation approach to dynamic semantics, because Java does not support higher-order type parametrization. One option to address this issue is to forbid such kinds of abstraction in ALGEBRA. Another option is to investigate some alternative approach to dynamic semantics. Yet another option is to still use a type-directed translation, but target a different calculus that supports higher-order type parametrization. We will explore these options. We will also extend our language support for object algebras. In particular, we will study language support for generalized object algebras and corresponding delegation-based semantics discussed in our encodings of object algebras [13]. All extensions will be formalized and proved type-safe.

4.2.2 Task 2: Compilation

We intend to study three different compilation schemes for ALGEBRA, which provide different trade-offs in terms of *modularity*, *efficiency* and *simplicity of implementation*. The first compilation scheme will be based on the type-directed translation into FGJ. This will provide an easy compilation scheme, which is modular but not necessarily efficient. As a second step we will study a specialization-based, whole-program compilation strategy. This compilation strategy will be efficient, but non-modular. Finally, we will investigate a compilation scheme which approximates the efficiency of whole-program compilation, while still being modular by directly generating bytecode or native code. The implementation of the ALGEBRA compiler and the investigation of compilation schemes will require a significant amount of research, as well as significant engineering effort. Therefore this will be the foundation for a second graduate student (Student B) work.

As preliminary work for this task we have shown how to apply the type-directed translation based on the object algebra encodings manually. We believe that this illustrates the feasibility of a general, and automated type-directed translation compilation scheme.

Task 2.1: Compilation via type-directed translation The easiest compilation method for ALGEBRA is to build on the encodings of object algebras, and use the same type-directed translation that is employed to provide a dynamic semantics for ALGEBRA. In the type-directed translation there is a one-to-one correspondence between the language constructs in ALGEBRA and the object algebra encodings in Java [11]. We illustrate the outcome of this compilation scheme using the EPL example in Figure 5. Interfaces in ALGEBRA are simply translated into interfaces in Java. Object algebra interfaces are translated into generic Java interfaces. For example,

$$\textbf{interface}\ \mathit{ExpAlg}\langle \mathit{Exp}\rangle\{\,\mathit{Exp}\,\mathit{Lit}\,(\mathit{Int}\,x);\}$$

would be the result of translating the algebra interface *ExpAlg* in Figure 5. Sorts such as *Exp* become type parameters and constructors are encoded as regular methods. Algebras are translated into classes that implement the (translated) object algebra interfaces. Figure 3 illustrates what the translation of the algebra *Eval* would be. It is worth noting the amount of boilerplate code that is required in the Java code. Datatype declarations are translated into interfaces. For example, the datatype declaration for *NegExp* is translated into:

$$\textbf{interface}\ \mathit{NegExp}\,\{\,\langle \mathit{Exp}\rangle\ \mathit{Exp}\ \ \mathit{apply}\,(\mathit{AllAlg}\langle \mathit{Exp}\rangle\ \mathit{alg});\}$$

A value such as exp_2 would be translated into a variable:

$$\begin{array}{l}\mathit{NegExp}\,\mathit{exp}_2 = \textbf{new}\,\mathit{NegExp}\,()\,\{\\ \quad \textbf{public}\langle \mathit{Exp}\rangle \mathit{Exp}\,\mathit{apply}\,(\mathit{AllAlg}\langle \mathit{Exp}\rangle\ \mathit{alg})\,\{\\ \quad\quad \textbf{return}\,\mathit{alg}.\mathit{Add}\,(\mathit{exp}_1.\mathit{apply}\,(\mathit{alg}),\mathit{alg}.\mathit{Neg}\,(\mathit{alg}.\mathit{Lit}\,(2)));\\ \quad \}\};\\ \};\end{array}$$

The most interesting part is the translation of the expression $exp_2\langle \mathit{NegPrint},\mathit{Eval}\rangle$. The first challenge is how to translate the intersection type of this expression $\mathit{IEval}\ \textbf{with}\ \mathit{IPrint}$. This is achieved by creating an interface on-the-fly, that implements both IEval and IPrint:

$$\textbf{interface}\,\mathit{IEvalwithIPrint}\ \ \textbf{extends}\ \ \mathit{IEval},\mathit{IPrint}\,\{\,\}$$

The second challenge is how to provide the algebra composed of *NegPrint* and *Eval*. This requires significant amount of boilerplate code, but the key idea is to have a class of the form:

$$\begin{array}{l}\textbf{class}\,\mathit{CombineNegPrintEval}\ \ \textbf{implements}\ \ \mathit{AllAlg}\langle \mathit{IEvalwithIPrint}\rangle\{\\ \quad \mathit{NegPrint}\,\mathit{alg1} = \textbf{new}\,\mathit{NegPrint}\,();\\ \quad \mathit{Eval}\,\mathit{alg2} = \textbf{new}\,\mathit{Eval}\,();\\ \quad \textbf{public}\,\mathit{IEvalwithIPrint}\,\mathit{Add}\,(\textbf{final}\,\mathit{IEvalwithIPrint}\,e_1,\textbf{final}\,\mathit{IEvalwithIPrint}\,e_2)\,\{\ldots\}\\ \quad \textbf{public}\,\mathit{IEvalwithIPrint}\,\mathit{Neg}\,(\textbf{final}\,\mathit{IEvalwithIPrint}\,e)\,\{\ldots\}\\ \quad \textbf{public}\,\mathit{IEvalwithIPrint}\,\mathit{Lit}\,(\textbf{final}\,\textbf{int}\,x)\,\{\ldots\}\\ \}\end{array}$$

This class uses the *NegPrint* and *Eval* algebras to delegate the behaviour of the constructors accordingly (see [13] for more details). Finally, $exp_2\langle \mathit{NegPrint},\mathit{Eval}\rangle$ translates into $exp_2.\mathit{apply}\,(\textbf{new}\,\mathit{CombineNegPrintEval}\,())$.

This strategy has a few important benefits, including: 1) simplicity of implementation; 2) support for modular type checking and separate compilation; 3) no need for language infrastructure (all works in a standard JVM); and 4) the compilation scheme directly corresponds to the dynamic semantics. However, there is also an important disadvantage: there is significant performance and memory overhead introduced by the abstractions used in the object algebra encodings in Java.

Task 2.2: Specialization-based Compilation To target the issues with performance and memory overhead, we will explore a specialization-based compilation scheme. It is well-known that modularity usually comes with some associated overhead. However, several works [69, 1, 54] have shown that often overhead can be removed by inlining definitions and using techniques similar to partial evaluation. We expect similar techniques to be helpful in ALGEBRA. The idea is to remove the overhead that arizes from compositions of object algebras by inlining the composition code. This would effectively compile a program written in ALGEBRA into corresponding (non-modular) versions of these programs written in Java. Consequently, this compilation method would achieve performance comparable to hand-written non-modular programs. In certain scenarios, where performance is a key concern, this compilation scheme may be the best. The main drawback will be the loss of separate compilation.

Task 2.3: Native Compilation To recover separate compilation and still have good performance and low memory overhead, we will study another compilation scheme that moves away from the strategy of generating Java code and generates bytecode/machine code directly. Existing OO languages and virtual machines provide highly optimized representations for objects and efficient composition via inheritance. However, the composition mechanisms for object algebras cannot be encoded by the existing inheritance mechanism, so they cannot benefit from that highly optimized mechanism. We need to find better low-level representations for object algebras, and most importantly efficient ways to perform composition. This means that significant new technology needs to be developed, including new run-time systems or virtual machines. Furthermore, this optimized low-level compilation scheme will be significantly different from the dynamic semantics of ALGEBRA. This will pose additional challenges in terms of showing equivalence between the type-directed translation based dynamic semantics and the corresponding compiled programs. Nevertheless, we believe that this will be a very interesting, but necessarily longer term, direction to promote ALGEBRA as a solid, and viable programming language alternative to existing SPL tools.

4.3 Task 3: Experimental Validation and Case Studies.

Finally to illustrate the applicability and benefits of ALGEBRA, we will develop several SPL case studies. These case studies will range from small, well-known canonical SPL examples, such as the EPL or the Graph Product Line [51], to larger real-world studies. In particular, for the larger case studies we want to build on our background on programming languages to develop a modular SPL for programming language components. This SPL can be used to reuse parts of interpreters, compilers, program analyzers, parsers and other programming language related infrastructure. The case studies will involve significant engineering effort and both graduate students will be involved in the development the case studies.

As preliminary work, we have already shown how to implement the canonical EPL product line in Figure 5. Of course, this is a tiny and toy example, and larger case studies need to be developed to validate the applicability of ALGEBRA for developing SPLs. Nevertheless, we believe that this case study already illustrates the benefits in terms of simplicity of ALGEBRA compared to other programming approaches like virtual types [57], or object algebras encodings in Java.

A Programming Language Case Study As a starting point for the larger programming language SPL, we will take the components for interpreters and corresponding type system that we have developed previously in the MTC and 3MT frameworks [33, 34]. MTC and 3MT can already be used to develop reusable infrastructure for small languages like Mini-ML. Our goal is to use ALGEBRA to implement reusable programming language components that will scale up to languages like Java. We believe ALGEBRA will provide an alternative to extensible compiler frameworks such as Polyglot [56], JastAdd [43] or JustAddJ [36]. These frameworks allow easily extending compilers with new features and can be viewed as a specialized form of SPL tools. Similarly to FeatureHouse or AHEAD, they take a meta-programming approach that does not support modular type-checking or separate compilation. Both Polyglot and JustAddJ have extensive and realistic case studies that would be interesting to port to ALGEBRA for comparison. The use of ALGEBRA should provide the obvious benefits of modular type-checking and separate compilation. Porting those case studies will also be helpful to expose limitations in terms of expressiveness and can provide us with feedback for useful extensions to ALGEBRA. Both Polyglot and JustAdd have several interesting modularity features, we expect that some of these would be interesting to investigate in the context of ALGEBRA.

Bibliography

[1] A. Alimarine and S. Smetsers. Improved fusion for optimizing generics. In *Proceedings of the 7th International Conference on Practical Aspects of Declarative Languages*, PADL'05, pages 203–218, 2005.

[2] S. Apel and C. Kästner. An overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84, 2009.

[3] S. Apel, C. Kästner, A. Grösslinger, and C. Lengauer. Type safety for feature-oriented product lines. *Automated Software Engg.*, 17(3):251–300, Sept. 2010.

[4] S. Apel, C. Kastner, and C. Lengauer. Featurehouse: Language-independent, automated software composition. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 221–231, 2009.

[5] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. FeatureC++: on the symbiosis of feature-oriented and aspect-oriented programming. In *Proceedings of the 4th International Conference on Generative Programming and Component Engineering*, GPCE'05, pages 125–140, 2005.

[6] S. Apel, T. Leich, and G. Saake. Aspectual mixin layers: aspects and features in concert. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 122–131, 2006.

[7] S. Apel and C. Lengauer. Superimposition: a language-independent approach to software composition. In *Proceedings of the 7th International Conference on Software Composition*, SC'08, pages 20–35, 2008.

[8] D. Batory. Feature-Oriented Programming and the AHEAD Tool Suite. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 702–703, 2004.

[9] D. Batory and B. J. Geraci. Composition Validation and Subjectivity in GenVoca Generators. *IEEE Trans. Softw. Eng.*, 23(2):67–82, Feb. 1997.

[10] **B. C. d. S. Oliveira**. Modular visitor components: A practical solution to the expression families problem. In *ECOOP '09: Proceedings of the 23rd European Conference on Object Oriented Programming*, 2009.

[11] **B. C. d. S. Oliveira** and W. R. Cook. Extensibility for the masses - practical extensibility with object algebras. In *ECOOP*, pages 2–27, 2012.

[12] **B. C. d. S. Oliveira** and W. R. Cook. Functional programming with structured graphs. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 77–88, 2012.

[13] **B. C. d. S. Oliveira**, T. V. der Storm, A. Loh, and W. R. Cook. Feature-oriented programming with object algebras. In *ECOOP*, 2013.

[14] **B. C. d. S. Oliveira** and J. Gibbons. Typecase: a design pattern for type-indexed functions. In *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, Haskell '05, 2005.

[15] **B. C. d. S. Oliveira** and J. Gibbons. Scala for generic programmers. *Journal of Functional Programming*, 20(Special Issue 3-4):303–352, 2010.

[16] **B. C. d. S. Oliveira**, R. Hinze, and A. Löh. Extensible and modular generics for the masses. In *Trends in Functional Programming*, 2006.

[17] **B. C. d. S. Oliveira**, A. Moors, and M. Odersky. Type classes as objects and implicits. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 341–360, 2010.

[18] **B. C. d. S. Oliveira**, T. Schrijvers, W. Choi, W. Lee, and K. Yi. The implicit calculus: a new foundation for generic programming. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 35–44, 2012.

[19] **B. C. d. S. Oliveira**, T. Schrijvers, and W. R. Cook. Effectiveadvice: disciplined advice with explicit effects. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development, AOSD 2010*, pages 109–120, 2010.

[20] **B. C. d. S. Oliveira**, T. Schrijvers, and W. R. Cook. MRI: Modular reasoning about interference in incremental programming. *Journal of Functional Programming*, 22(06):797–852, 2012.

[21] **B. C. d. S. Oliveira**, M. Wang, and J. Gibbons. The visitor pattern as a reusable, generic, type-safe component. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, 2008.

[22] E. Bodden, T. Tolêdo, M. Ribeiro, C. Brabrand, P. Borba, and M. Mezini. SPLLIFT: statically analyzing software product lines in minutes instead of years. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, PLDI '13, pages 355–364, 2013.

[23] C. Böhm and A. Berarducci. Automatic synthesis of typed lambda-programs on term algebras. *Theor. Comput. Sci.*, 39:135–154, 1985.

[24] C. Böhm and G. Jacopini. Flow diagrams, turing machines and languages with only two formation rules. *Commun. ACM*, 9(5):366–371, May 1966.

[25] M. Böhme, **B. C. d. S. Oliveira**, and A. Roychoudhury. Partition-based regression verification. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, 2013.

[26] M. Böhme, **B. C. d. S. Oliveira**, and A. Roychoudhury. Regression tests to expose change-interaction errors. In

Proceedings of the 2013 Joint meeting of ACM SIGSOFT symposium and European conference on Foundations of software engineering, ESEC/FSE '13, 2013.

[27] K. B. Bruce, M. Odersky, and P. Wadler. A statically safe alternative to virtual types. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, ECCOP '98, pages 523–549, 1998.

[28] P. Canning, W. Cook, W. Hill, W. Olthoff, and J. C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proceedings of the fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 273–280.

[29] P. Clements and L. Northrop. *Software product lines: practices and patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[30] M. Coppo and M. Dezani-Ciancaglini. A new type assignment for λ-terms. *Archiv fr mathematische Logik und Grundlagenforschung*, 19(1):139–156, 1978.

[31] O.-J. Dahl and K. Nygaard. Simula: An ALGOL-based simulation language. *Communications of the ACM*, 9(9):671–678, 1966.

[32] F. Damiani, L. Padovani, and I. Schaefer. A formal foundation for dynamic delta-oriented software product lines. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*, GPCE '12, pages 1–10, 2012.

[33] B. Delaware, **B. C. d. S. Oliveira**, and T. Schrijvers. Meta-theory à la carte. In *POPL '13: Proceedings of the 40th ACM SIGPLAN-SIGACT Conference on Principles on Programming Languages*, 2013.

[34] B. Delaware, S. Keuchel, T. Schrijvers, and **B. C. d. S. Oliveira**. Modular monadic meta-theory. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, 2013.

[35] E. W. Dijkstra. Letters to the editor: go to statement considered harmful. *Commun. ACM*, 11(3):147–148, Mar. 1968.

[36] T. Ekman and G. Hedin. The JastAdd extensible Java compiler. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, OOPSLA '07, pages 773–774, 2007.

[37] E. Ernst. Family polymorphism. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 303–326, 2001.

[38] E. Ernst, K. Ostermann, and W. R. Cook. A virtual class calculus. In *Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 270–282, 2006.

[39] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[40] J. Gibbons and **B. C. d. S. Oliveira**. The essence of the iterator pattern. *J. Funct. Program.*, 19(3-4), July 2009.

[41] J. V. Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, WICSA '01, 2001.

[42] W. Harrison and H. Ossher. Subject-oriented programming: a critique of pure objects. In *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '93, pages 411–428, 1993.

[43] G. Hedin and E. Magnusson. JastAdd: an aspect-oriented compiler construction system. *Sci. Comput. Program.*, 47(1):37–58, Apr. 2003.

[44] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001.

[45] C. Kastner, S. Apel, and D. Batory. A Case Study Implementing Features Using AspectJ. In *Proceedings of the 11th International Software Product Line Conference*, SPLC '07, pages 223–232, 2007.

[46] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, 2001.

[47] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. marc Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP '97: European Conference on Object Oriented Programming*. SpringerVerlag, 1997.

[48] C. H. P. Kim, D. S. Batory, and S. Khurshid. Reducing combinatorics in testing product lines. In *Proceedings of the tenth International Conference on Aspect-oriented Software Development*, AOSD '11, pages 57–68, 2011.

[49] B. Liskov and S. Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN symposium on Very high level languages*, pages 50–59, 1974.

[50] R. E. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating support for features in advanced modularization technologies. In *Proceedings of the 19th European conference on Object-Oriented Programming*, ECOOP'05, pages 169–194, 2005.

[51] R. E. Lopez-Herrejon and D. S. Batory. A standard problem for evaluating product-line methodologies. In *Proceedings of the Third International Conference on Generative and Component-Based Software Engineering*, GCSE '01, pages 10–24, 2001.

[52] R. E. Lopez-Herrejon, D. S. Batory, and W. R. Cook. Evaluating support for features in advanced modularization technologies. In *ECOOP '05*, pages 169–194, 2005.

[53] D. MacQueen. Modules for standard ML. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, LFP '84, pages 198–207, 1984.

[54] J. P. Magalhães. Optimisation of generic programs through inlining. In *In 24th Symposium on Implementation and Application of Functional Languages (IFL'12)*, IFL '12, 2013.

[55] M. Mezini and K. Ostermann. Variability management with feature-oriented programming and aspects. In *Proceedings of the 12th ACM SIGSOFT twelfth International Symposium on Foundations of Software Engineering*, SIGSOFT '04/FSE-12, pages 127–136, 2004.

[56] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: an extensible compiler framework for java. In *Proceedings of the 12th International Conference on Compiler Construction*, CC'03, pages 138–152, 2003.

[57] M. Odersky and M. Zenger. Independently Extensible Solutions to the Expression Problem. In *Proc. of 12th Intl. Workshop on Foundations of Object-Oriented Languages*, Jan. 2005.

[58] K. Pohl, G. Böckle, and F. J. v. d. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[59] C. Prehofer. Feature-oriented programming: A fresh look at objects. In *ECOOP '97: European Conference on Object Oriented Programming*, pages 419–443. Springer, 1997.

[60] C. Saito, A. Igarashi, and M. Viroli. Lightweight family polymorphism*. *J. Funct. Program.*, 18(3):285–331, May 2008.

[61] I. Schaefer, L. Bettini, and F. 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, 2011.

[62] T. Schrijvers and **B. C. d. S. Oliveira**. Monads, zippers and views: virtualizing the monad stack. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 32–44, 2011.

[63] T. Sheard. Accomplishments and research challenges in meta-programming. In *In 2nd Int. Workshop on Semantics, Applications, and Implementation of Program Generation, LNCS 2196*, pages 2–44. Springer-Verlag, 2000.

[64] Y. Smaragdakis and D. Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Methodol.*, 11(2), Apr. 2002.

[65] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 107–119, 1999.

[66] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *Proceedings of the 6th International Conference on Generative Programming and Component Engineering*, GPCE '07, pages 95–104, 2007.

[67] M. Torgersen. The expression problem revisited four new solutions using generics. In *In Proceedings of the 18th European Conference on Object-Oriented Programming*, pages 123–143. Springer-Verlag, 2004.

[68] M. VanHilst and D. Notkin. Using role components in implement collaboration-based designs. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, pages 359–369, 1996.

[69] T. L. Veldhuizen. C++ templates as partial evaluation. In *PEPM*, pages 13–18, 1999.

[70] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Principles of Programming Languages*, pages 60–76. ACM, Jan. 1989.

[71] P. L. Wadler. The expression problem. Posting to java-genericity mailing list, 12th Nov 1998.

Homogeneous
similar code in multiple locations
examples: logging, profiling, security

```
interface Exp {String print();}

class Lit(x : Int) extends Exp {
  public print() {
    println("Entering print()");
    return x.toString();
  }
}
class Add(e1 : Exp, e2 : Exp) extends Exp {
  public print() {
    println("Entering print()");
    return e1.print() + " + " + e2.print();
  }
}
```

Heterogeneous
different code in multiple locations
example: adding features to a compiler, adding new methods to an existing class hierarchy

```
interface Exp {
  Int eval();
  String print();
}

class Add(e1 : Exp, e2 : Exp) extends Exp {

  public eval() {return e1.eval() + e2.eval();}

  public print() {
    return e1.print() + " + " + e2.print() + " = " + e2.eval();
  }
}
```

Figure 1: Examples of different types of crosscutting concerns.

Program	Operations		Data types		
	Print	Eval	Lit	Add	Neg
1	✓		✓		
2	✓	✓	✓		
3	✓		✓	✓	
4	✓	✓	✓	✓	
5	✓		✓		✓
6	✓	✓	✓		✓
7	✓		✓	✓	✓
8	✓	✓	✓	✓	✓

Figure 2: Possible combination of features in the EPL.

```
class Eval implements AddAlg⟨IEval⟩{
  public IEval Lit (final int x) {
    return new IEval () {
      public int eval () {return x;}
    };
  }
  public IEval Add (final IEval e1 ,final IEval e2) {
    return new Eval () {
      public int eval () {return e1 .eval ()+e2 .eval ();}
    };
  }
}
```

Figure 3: Object Algebra for Evaluation in Java.

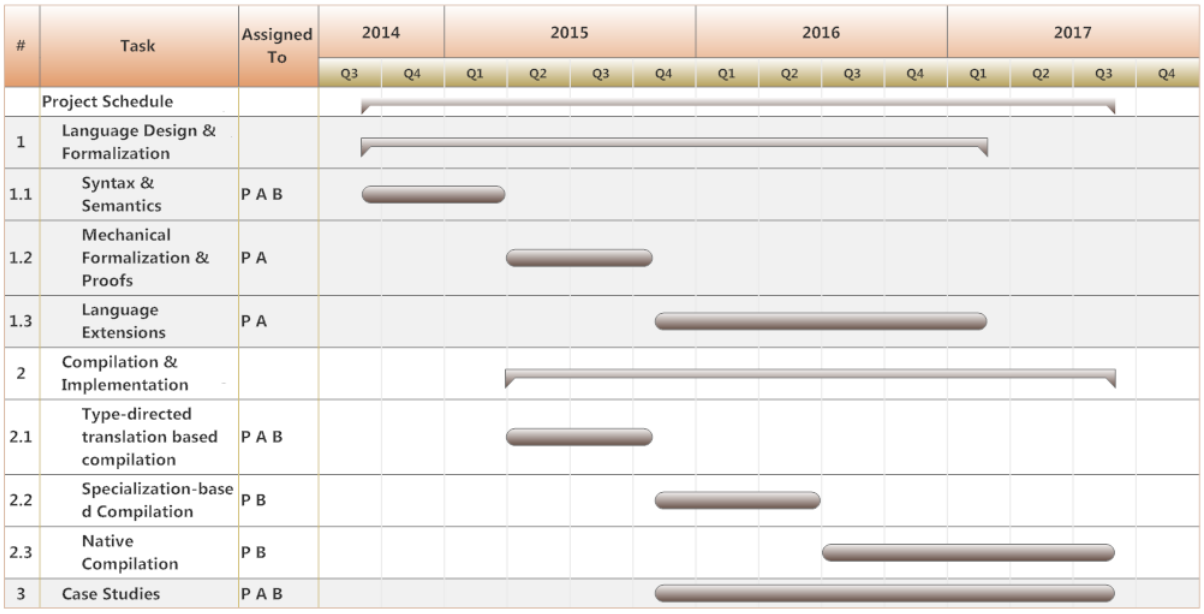

Figure 4: Project Schedule

```
   -- (Feature) interfaces
interface IEval where
   eval:Int
interface IPrint where
   print:String
   -- Algebra interfaces
algebra interface ExpAlg where
   sort Exp
   Lit  :Int → Exp
algebra interface AddAlg  extends  ExpAlg where
   Add:Exp → Exp → Exp
algebra interface NegAlg  extends  ExpAlg where
   Neg :Exp → Exp
algebra interface AllAlg  extends  AddAlg,NegAlg
   -- Algebras: Printing and Evaluation
algebra Print  implements  AddAlg where
   sort Exp            =IPrint
   print@(Lit x)       =x.toString
   print@(Add e1 e2)=e1.print+" + "+e2.print
algebra NegPrint  extends  Print  implements  AllAlg where
   print@(Neg e)       ="-"+e.print
algebra Eval  implements  AllAlg where
   sort Exp            =IEval
   eval@(Lit x)        =x
   eval@(Add e1 e2)  =e1.eval+e2.eval
   eval@(Neg e)        =-e.eval
   -- Datatypes from algebras
data Exp  from  AddAlg.Exp
data NegExp  from  AllAlg.Exp
   -- Values
exp1:Exp
exp1=Add (Lit 3) (Lit 4)
exp2:NegExp
exp2=Add exp1 (Neg (Lit 2))
   -- Main Program
main=
   let o1:IPrint=exp1⟨Print⟩ in
   let o2:IEval with IPrint=exp2⟨NegPrint,Eval⟩ in
   println("First expression: "+o1.print)
   println("Second expression: "+o2.print+" evaluates to: "+o2.eval)
```

Figure 5: Implementing the Expression Product Line in ALGEBRA.