FOAL '07

Proceedings of the Sixth Workshop on Foundations of Aspect-Oriented Languages

held at the
Sixth International Conference on
Aspect-Oriented Software Development

March 12–16, Vancouver, British Columbia, Canada

Workshop Organizers: Curtis Clifton, Gary T. Leavens, and Mira Mezini

ACM International Conference Proceedings Series
ACM Press

The Association for Computing Machinery
1515 Broadway
New York New York 10036

ACM ISBN: 1-59593-671-4

Contents

Preface

Aspect-oriented programming is a paradigm in software engineering and programming languages that promises better support for separation of concerns. The sixth Foundations of Aspect-Oriented Languages (FOAL) workshop was held at the Sixth International Conference on Aspect-Oriented Software Development in Vancouver, Canada, on March 13, 2007. This workshop was designed to be a forum for research in formal foundations of aspect-oriented programming languages. The call for papers announced the areas of interest for FOAL as including: semantics of aspect-oriented languages, specification and verification for such languages, type systems, static analysis, theory of testing, theory of aspect composition, and theory of aspect translation (compilation) and rewriting. The call for papers welcomed all theoretical and foundational studies of foundations of aspect-oriented languages.

The goals of this FOAL workshop were to:

- Make progress on the foundations of aspect-oriented programming languages.
- Exchange ideas about semantics and formal methods for aspect-oriented programming languages.
- Foster interest within the programming language theory and types communities in aspect-oriented programming languages.
- Foster interest within the formal methods community in aspect-oriented programming and the problems of reasoning about aspect-oriented programs.

FOAL logos courtesy of Luca Cardelli

The workshop was organized by Curtis Clifton (Rose-Hulman Institute of Technology), Gary T. Leavens (Iowa State University), and Mira Mezini (Darmstadt University of Technology). The program committee was chaired by Shmuel Katz (Technion–Israel Institute of Technology).

We thank the organizers of AOSD 2007 for hosting the workshop, and Workshops Chairperson William Harrison in particular for his help with Digital Library publication of these proceedings.

Message from the Program Committee Chair

FOAL has become one of the primary venues for work on the formal foundations of aspect languages. The reviewing process for FOAL2007 was, as usual, detailed and thorough, beyond what is typical in a Workshop. Every paper was reviewed by at least three, and generally four, reviewers, who provided constructive criticism and valuable feedback for the authors. The promptness and efforts of the reviewers are greatly appreciated, and have again allowed us to construct an interesting FOAL program. The papers provide a cross-section of work on formal methods and semantics for aspects, from one on refinement for aspects, to papers on type systems for aspects, treatment of dynamic aspects, connections to nonmonotonic logic and monads, and issues in bytecode slicing for aspects.

The members of the program committee were: Curtis Clifton (Rose-Hulman Institute of Technology), Rémi Douence (Ecole des Mines de Nantes, Inria, Lina), Pascal Fradet (INRIA), Stephan Herrmann (Technische Universität Berlin), Alan Jeffrey (Bell Labs), Shmuel Katz (Technion–Israel Institute of Technology), Ralf Lämmel (Microsoft) Gary Leavens (Iowa State University), Karl Lieberherr (Northeastern University), David Lorenz (University of Virginia), Todd Millstein (University of California, Los Angeles), Mira Mezini (Darmstadt University of Technology), James Riely (DePaul University), and Mitchell Wand (Northeastern University).

The sub-reviewers, whom we also thank, were: Ahmed Abdelmeged, Bryan Chadwick, Christine Hang, and Therapon Skotiniotis.

I would also like to warmly thank the organizing committee of FOAL, Curtis Clifton, Gary Leavens, and Mira Mezini, for their untiring work in bringing together the various elements needed to create a vibrant workshop.

Shmuel Katz
FOAL '07 Program Chair
Technion–Israel Institute of Technology

Fundamentals of Concern Manipulation

Talk Abstract

Harold Ossher
IBM T. J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10590, USA
+1-914-784-7975
ossher@us.ibm.com

ABSRACT

This talks describes a number of principles and key concepts underlying *concern manipulation*, the use of concerns to aid in a variety of software development tasks. Concern modeling and exploration, query and composition are considered. The principles and concepts guided work on the Concern Manipulation Environment (CME), which provides both prototype tools supporting aspect-oriented software development, and flexible components for use in building such tools.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features, D.2.3 [**Software Engineering**]: Coding Tools and Techniques, D2.13 [**Software Engineering**]: Reusable Software, D.2.2 **[Software Engineering**]: Design Tools and Techniques.

General Terms

Languages, Design.

Keywords

Aspect-oriented software development, separation of concerns, software queries, software decomposition and composition.

1. INTRODUCTION

As its name suggests, *concern manipulation* is about the use of concerns in any and all ways that are useful. This includes:

- Writing software that is modularized by concern.
- Identifying or mining concerns that were not modularized
- Modeling concerns and their relationships, and using these models to aid in development activities, such as assessing impact of change.
- Extracting concerns that are tangled with others.
- Composing concerns in flexible ways to yield full systems.

Workshop FOAL '07, March 12-13, 2007 Vancouver, BC, Canada.

This talk is about general principles and key concepts of concern manipulation.

The principles were key considerations in the design and implementation of the Concern Manipulation Environment (CME) [5]. It provides both a set of prototype tools and a set of flexible components. The tools are for use during aspect-oriented development, and include a Concern Explorer for navigating and populating an underlying concern model [4], a query tool for searching for software elements using a variety of attributes and relationships [8], and a composition tool for composing concerns as guided by high-level, mostly simple specifications. The components are for tool builders to build upon and researchers to use for experimentation and prototyping. They include components for concern modeling [4], query [8], composition [7, 2] and related sub-activities. Extraction was planned but not implemented. The components are general and flexible, intended to be tailorable to a variety of AOSD approaches applied to a variety of different types of artifacts.

The CME is an open source project, though not currently under active development. It was developed as an Eclipse Technology Project, and is now available on SourceForge [1].

The rest of this abstract merely lists the principles and concepts covered, or alluded to, in the talk. In a few cases, it identifies architectural implications for tools aimed at supporting general concern manipulation. Further explanation and details, as well as discussion of and references to related work, are available in the referenced publications.

2. PRINCIPLES AND KEY CONCEPTS

This section begins with some general principles and concepts, and then discusses concerns, query and composition in separate subsections.

- The various concern-manipulation tools and components should provide a unified view and experience. This implies sharing of concepts wherever possible, such as regarding the body of software being worked on.
- The body of software being worked on is in a u*niverse* consisting of *container spaces* of *containers* made up of *elements.*
    - In the important special case of object-oriented software, the container spaces are *type spaces* (e.g., Java class paths), the con-

tainers are *types* (e.g., classes and interfaces), and the elements are *members* (methods and fields).

  - Containers that are referenced but are not to be manipulated themselves, such as Java library classes in most contexts, can be included in a special *library* container space, which is considered to be included in all container spaces.
  - All names used within a container space must be uniquely defined within that space (perhaps in the automatically-included library space). This is necessary for names to be properly understood and processed.

- The universe can, and usually does, involve software artifacts/elements of various kinds.
  - Architectural implication: Artifact-kind-specific code should be isolated, so that the bulk of the concern-manipulation support is generic.

- *Methoid*: a pattern identifying material inside element bodies, allowing the matching material to be treated as extractable methods for the purpose of identification, searching and composition. This allows support for code-level join points such as calls, throws and exception handler bodies.
- *Correspondence:* a tuple of *corresponding entities* (container spaces, containers, elements or methoids) that are to be composed with one another to form a composed entity. Correspondences identify join points in a symmetric way, and correspondence queries are the symmetric analogy of pointcuts.
- Each entity has *attributes,* which can be used in queries. When corresponding entities are composed, their attributes must be combined. Attributes include:
  - *Modifiers:* keyword attributes, e.g., "public."
  - *Classifiers:* modifiers that serve to classify their entities, e.g., "interface."

2.1 Concerns

- Concerns should be first-class entities, explicitly represented (modeled) and manipulable by users and tools.
- An underlying *symmetric* model should be used, with a convenient *asymmetric* façade available. Both symmetric and asymmetric scenarios [6] are important: some concerns are naturally peers, possibly freestanding, whereas others are naturally extensions or specializations of *base* concerns. This approach provides convenient, unified support for both. It is possible because asymmetric models are restrictions of symmetric models.
- An individual concern can be heterogeneous, involving artifacts/elements of multiple kinds.
- A concern has an *intension*, indicating the meaning of the concern, and an *extension*, the set of software elements that currently pertain to it. The intension might be expressed by a query. In the degenerate case, the intension can be merely a comment and the extension specified explicitly.
- Software can be written explicitly encapsulated in concerns, such as in modules or packages that represent concerns. Concerns can also be obtained by identifying or mining elements scattered across other concerns.
- A concern, unlike a container space, may contain names that resolve to definitions not included in the concern. In general, obtaining container spaces from concerns requires *extraction*, which must deal with such names (perhaps by including definitions, or *requires* declarations, within the space).

2.2 Query

- Queries are needed in many contexts, such as for exploration, definition of concern intensions, and correspondence identification for composition.
- Uniform query support should be available in all contexts, and the same query language(s) be usable throughout.
- Different query languages and underlying engines are appropriate for different AOSD approaches and experiments.
  - Architectural implication: Query languages and engines should be extensible and pluggable.
- Despite this variation, to provide the uniform support desired, a query language must provide at least the following capabilities:
  - Selection of elements based on names (including parameter signatures for methods), modifiers, classifiers, attributes and containment.
  - Selection of methoids, based on their patterns.
  - Selection of relationships, based on their names and characteristics of their end points.
  - Selection of correspondences: tuples of corresponding elements related as desired (e.g., having the same unqualified names in different scopes).
  - Navigation via relationships, including transitive closure.
  - Predicates and set operations.
  - Variables and unification. This is absolutely required for correspondence queries used for composition, and us useful in other contexts also.

2.3 Composition

- *Static composition* is sufficient to support dynamic join points and pointcuts. *Dynamic residue,* where the paradigm requires runtime tests (or other activities) to be

performed at join points during execution, is handled by generating code to perform the appropriate tests and composing it statically at the right locations. This is, in fact, what aspect compilers typically do.

- Three composition levels are important, with different needs and tradeoffs:
    - *Concern assembly* level: the lowest level, at which the key issue is the nitty-gritty details of composing specific artifacts, such as Java class files.
    - *Reusable component* level: the middle level, at which the key issue is providing tool builders with flexible alternatives, allowing them to realize different composition paradigms.
    - *Tool* level: the highest level, at which the key issue is providing AOSD developers with convenient language constructs that support a particular paradigm.

2.3.1 Concern Assembly

Concern assembly involves some concepts specific to the low-level details of synthesizing composed artifacts from source artifacts:

- *Mapping* and *translation*, enabling a formal element, such as a method body, to be copied correctly from its source context to the composed context with proper name resolution.
- *Relationships* among elements, such as subtyping.
- *Method combination graphs,* specifying the details of how multiple, corresponding methods should be combined, including such issues as sequencing, exception handling and parameter mapping.
- Primitives for:
    - Container and element creation.
    - Mapping and relationship specification.
    - Copying and translating formal elements.
    - Generating code based on method combination graphs.

2.3.2 Reusable Composition Component

The CME composition component provides great flexibility by allowing composition to be specified in terms of the following concepts:

- *Weaving directive*s specify composition details.
- *What* elements are to be joined: correspondences.
- *How* elements are to be joined:
    - *Selection*, indicating which are to be included.
    - *Ordering,* specified by *combination graphs*.
    - *Structure,* specifying how the component elements are to be related in the composed result (e.g., facets of the same object, separate objects, separate object and aspect, etc.) [3].
- Making assumptions explicit:
    - *Encapsulation* indicates at what level name-matching is to be applied, if at all.
    - *Opacity* indicates whether class hierarchy structure is to be taken into consideration during composition, or if all classes are to be "flattened" before composition by having their inherited members explicitly included.
- Resolving multiple weaving directives that apply to the same element:
    - *Exclusivity* indicates whether multiple directives can cooperate to produce a single composed result, or whether just one must be selected.
    - *Precedence* determines the order of selection.

2.3.3 Tool-level composition

The concepts at the tool level are dependent on the paradigms (aspect languages or approaches) being implemented: the whole intent is that each tool be able to provide its own model and concepts. There is thus great variation at this level, but the following general concepts apply:

- Ideally, a composition tool should provide composition capabilities that are convenient and easy to understand. It need not necessarily provide the full flexibility of the lower levels, which are intended to be able to support multiple paradigms.
- Concerns should be first-class elements in composition specifications.
    - In general, obtaining container spaces needed for the lower levels of composition from concerns requires extraction, as noted earlier.
- For full integration with concern modeling, the composition specifications should be expressed as *composition relationships* between elements of the concern model.
- The composition specifications supported by the tool should be compiled down to the directives offered by the reusable composition component.
- *Dynamic residues* are handled at the tool level, since their details are paradigm-specific. The tool should generate methods that perform the desired runtime tests or other activities, together with directives causing the composition component to include them where appropriate.
- An *attribute rewriting system*, capable of transforming attributes of high-level composition specifications to those of mid-level weaving directives can provide some generic support for implementing diverse composition paradigms. The transformation is based on rules that (partially) define the paradigm.

3. CONCLUSION

This abstract described a number of principles and key concepts of concern manipulation. They were used in the design and implementation of the CME, but validation is limited due to the

limited number of tools built on the CME and limited experience obtained with them.

Follow-on research is an open area, including: validation and improvement of these concepts, exploration of alternatives and of design and implementation details, implementation of varied AOSD paradigms in terms of them, and exploration of new issues, such as handling of concerns containing artifacts that are versioned in an SCM system.

4. ACKNOWLEDGMENTS

The principles and concepts listed here were the product of joint work with Bill Harrison and Peri Tarr. Many of them derive from earlier work, by us or others. The CME was designed and implemented by a joint team from the IBM T.J. Watson Research Center and IBM Hursley Park. People involved at various stages of the project were: Matthew Chapman, Bill Chung, Andrew Clement, Adrian Colyer, Bill Harrison, Helen Hawkins, Sian January, Vincent Kruskal, Harold Ossher, Tova Roth, Stanley Sutton, Peri Tarr, and Frank Tip,

5. REFERENCES

This abstract refers only to our own detailed publications about the CME and the underlying concepts. Discussion of and references to related work can be found in each of them.

[1] CME web site: http://sourceforge.net/projects/cme/.

[2] William Harrison, Vincent Kruskal, Harold Ossher, Peri Tarr and Frank Tip, “Common Low-Level Support for Composition and Weaving.” OOPSLA ’02 Workshop on Tools for Aspect-Oriented Software Development.

[3] William Harrison and Harold Ossher, “Member-Group Relationships Among Objects.” AOSD ’02 Workshop on Foundations Of Aspect-Oriented Languages (FOAL).

[4] William Harrison, Harold Ossher, Stanley M. Sutton Jr., and Peri Tarr, “Concern Modeling in the Concern Manipulation Environment,” ICSE ’05 workshop on Modeling and Analysis of Concerns in Software (MACS ’05).

[5] W. Harrison, H. Ossher, S. Sutton, P. Tarr, “The Concern Manipulation Environment – Supporting Aspect-Oriented Software Development.” IBM Systems Journal 44(2): 309--318, 2005, special issue on Open Source Software.

[6] William Harrison, Harold Ossher and Peri Tarr, “Asymmetrically vs. Symmetrically Organized Paradigms for Software Composition.” Research Report RC22685, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, December, 2002.

[7] William Harrison, Harold Ossher and Peri Tarr, “General Composition of Software Artifacts.” In Proceedings of the 5th International Symposium on Software Composition (SC ’06), March 2006, Springer, LNCS 4089.

[8] Peri Tarr, William Harrison, and Harold Ossher, “Pervasive Query Support in the Concern Manipulation Environment.” IBM Research Report RC23343, 2005.

Requirement Enforcement by Transformation Automata

Douglas R. Smith
Kestrel Institute
3260 Hillview Avenue
Palo Alto, California 94304 USA
smith@kestrel.edu

ABSTRACT

The goal of this work is to treat safety and security policies as requirements to be composed in an aspectual style with a developing application. Policies can be expressed either logically or by means of automata. We introduce the concept of *transformation automaton*, which is an automaton whose transitions are labeled with program transformations. A transformation automaton is applied to a target program by a sound static analysis procedure. The effect is to perform a global transformation that enforces the specified policy. The semantic effect of this global transformation is explored.

In previous work we discussed how the intent of an AspectJ-style aspect can be expressed precisely and abstractly as a state invariant. Here, this result is generalized to handle invariants that are conditional and stated over both events and state properties. A policy stated in such a logical format can be translated to a transformation automaton that enforces it in a target program. The translation process is defined by a collection of inference schemes that can be mechanically instantiated and then solved, at least partially automatically, by deductive calculations.

1. INTRODUCTION

This paper takes steps toward a deep integration of two worlds - the burgeoning field of Aspect-Oriented Software Development (AOSD) and the field of formal software development by mechanized refinement. These two fields have much to offer each other. Viewing each from the point of view of the other provides insights leading to cross-fertilization and new generalizations of both.

Formal software development starts with real-world requirements that are formalized into specifications. Specifications are then subjected to a series of refinements that preserve properties while introducing implementation details. Most work on development-by-refinement takes a posit-and-prove approach: a refinement is manually written that adds implementation detail to the current design specification, and the refinement is proved correct on the side. In contrast, our work has focused on generating refinements by applying representations of abstract design knowledge and using automated reasoning [20, 21]. To achieve wider acceptance and lower lifecycle costs, it is necessary to develop highly automated means for generating refinements.

A crucial fact of complex system design is that no matter how one designs the hierarchical structure of a system, there are always concerns that cross-cut the component structure and introduce dependencies that are not exposed at the component interfaces. These dependencies complicate the understanding and evolution of the system. We view cross-cutting concerns (such as aspects, safety and security policies, nonfunctional requirements) as behavioral requirements on a system.

The main contribution of AOSD is the development and popularization of means for expressing these cross-cutting requirements, or at least implementation prescriptions for them, in modular syntax, and providing automatic methods for weaving or composing them into one's design. What has been lacking is means for specifying the intent of aspects. In previous work [22] we showed how many AspectJ-style aspects can be specified by means of state invariants, and how aspect weaving can be performed as invariant maintenance. By starting with a logical specification of the intent of a cross-cutting concern, we showed how to derive what are called the pointcuts and advice of AspectJ aspects [10]. The derivation process provides assurance that the joinpoints are complete and that the advice correctly implements the specification.

This paper continues our focus on abstract, yet precise means for specifying the intent of cross-cutting requirements. Our previous results are generalized to handle invariants that are conditional and stated over both events and state properties. In particular, the specification of safety and security policies typically requires taking behavioral context into account when deciding whether current actions are acceptable. Policies can be expressed either logically or by means of automata, as convenient.

To implement cross-cutting requirements, we introduce the concept of *transformation automata*, which are automata whose transitions are labeled with program transformations. A transformation automaton is applied to a target program by a sound static analysis procedure. The effect is to per-

Sixth International Workshop on Foundations of Aspect-Oriented Languages (FOAL 2007), March 13, 2007, Vancouver, BC, Canada.

form a global transformation that enforces the specified policy by applying a collection of local transformations. We show how to calculate transformation automata from specifications of cross-cutting requirements.

Formal development poses several questions. What is the semantic effect of mechanically composing a cross-cutting requirement into a program? Is the requirement correctly and completely realized? Do previously satisfied requirements remain satisfied? We examine these issues in the context of a variety of examples.

The goal of this work is to treat cross-cutting concerns as requirement specifications, and to introduce the broadest possible range of mechanisms for composing/weaving those cross-cutting concerns in the context of a refinement process that generates correct-by-construction code. After introduction of notations, we work through a series of examples.

2. PRELIMINARIES

A behavior of a program can be represented graphically as a trace of alternating states and actions

$$state_0 \xrightarrow{act_0} state_1 \xrightarrow{act_1} state_2 \xrightarrow{act_2} state_3 \cdots$$

or more formally as a sequence of *transition triples* of the form

$$\langle state_i, act_i, state_{i+1} \rangle,$$

where states are a mapping from variables to values, and actions are state-changing operations (i.e. program statements). If x is a state variable and s a state, then $s.x$ denotes the value of x in s. Further, in the context of the transition triple $\langle state_0, act, state_1 \rangle$, x will refer to the value of x in the preState, $state_0.x$, and x' refers to the value in the postState, $state_1.x$.

For concreteness, an action is represented by abstract syntax so that we can perform pattern-matching and other syntactical operations and tests. The following operators construct sequences, including traces: *nil*, written $[]$, and $append(S, a)$, written $S :: a$ for sequence S and element a.

The semantics of a system S is given by a set of traces $Traces(S)$. To specify a system, we determine the observations that a stakeholder could make, and then write constraints on the observable state properties and event orderings. Here we assume that the observables of the system are exactly the states and actions of a trace; e.g. we cannot observe the state while a (primitive) action is taking place.

Actions are specified in a pre- and post-condition style. For example, the specification

assume: $x \geq 0$
achieve: $x' * x' = x \ \wedge \ x' \geq 0$

is satisfied by the action $x := \sqrt{x}$.

A *refinement* is a morphism in a suitable category of specifications. Intuitively, a refinement morphism preserves structure and properties. For algebraic specifications, a refinement morphism maps vocabulary such that typing is preserved, and formulas/sentences remain provable under translation (i.e. theorems are preserved). This means that properties are preserved. For behavioral specifications, a refinement morphism maps vocabulary such that typing is preserved, theorems are preserved, and domain behavior is simulated by codomain behavior [15]. The last condition implies that if system S refines to system T then $Traces(T) \subseteq Traces(S)$, or more generally that there is a simulation map from traces of T to traces of S.

Reification

In order to specify requirements that express cross-cutting features, we often need to reify certain extra-computational values such as history, the runtime call stack, the runtime heap, or external agents.

Suppose for example that we need some way to discuss the history of the program at any point in time. The execution history of the program can be reified into the state by means of a *specification* variable (sometimes called a shadow or ghost variable). That is, imagine that with each action taken by the program there is a concurrent action to update a variable called *hist* that records the history up until the current state; so each transition has the form

$$\langle st_i, \ (act_i \ || \ hist \ := \ hist :: \langle st_i, act_i, st_{i+1} \rangle), \ st_{i+1} \rangle$$

where $\alpha || \beta$ denotes parallel composition of actions α and β. Obviously this would be an expensive variable, but it is only needed for specification purposes, and typically at most a residue of it will appear in the executable code.

Other common examples of values to reify include the call stack (to constrain dynamic control context), heap (to constrain dynamic data context), time (to state performance constraints), and agency (to express the principals who are responsible for system actions).

3. EXAMPLE: AUTOSAVE REQUIREMENT

Suppose that we are developing a data editing application, and we desire to impose an autosave requirement (adapted from [1]): every 6 changes to the data from a file, save the data back out. With the aid of the reified variable *hist*, a specification of this requirement is easily stated:

$\Box \ cnt = (length \cdot dataop? \triangleright action \star hist) \ mod \ 6$
$\Box \ cnt = 5 \implies data = file$

where
(1) the *action* function selects the action from a transition $\langle state_i, act_i, state_{i+1} \rangle$
(2) $\star$ is the image operator, so $action \star hist$ is the list of actions performed up to the present
(3) *dataop*? holds for the representation of an action that changes the data of concern
(4) $\triangleright$ is the filter operator, so $dataop? \triangleright action \star hist$ is the list of dataops performed up to the present
(5) $\Box$ the always modality of temporal logic [12]; $\Box\phi$ asserts that the state (or transition) formula ϕ holds invariantly at every state of a trace.
In words, the two formulas assert that in every observable state, the variable *cnt* records the number of dataops modulo 6 that have occurred to that point in the current behavior (which is recorded in *hist*), and furthermore, in each state in which *cnt* has value 5, the data and the file have the same contents.

3.1 Establishing the Invariant

We have two invariants to establish, and we proceed along the lines presented in [22], by simultaneously deriving the essential parts of a inductive proof and the transformations that carry them out.

The first step is to generate code to establish the invariant initially, by satisfying the following two specifications:

assume: $hist = []$
achieve: $cnt' = lengthDataops \ mod \ 6$

where we abbreviate

$$length \cdot dataop? \triangleright action \star hist$$

by *lengthDataops*. The postcondition can be simplified as follows:

$cnt' = (length \cdot dataop? \triangleright action \star hist) \ mod \ 6$

$\iff$ { using the definition of *hist* and simplifying }

$cnt' = 0$

which is satisfied by the initialization code

$$cnt \ := \ 0.$$

Generating initialization code for the other invariant is similar:

assume: $hist = [] \ \wedge \ cnt = 0$
achieve: $cnt = 5 \implies data = file$

The postcondition can be simplified as follows:

$cnt = 5 \implies data = file$

$\iff$ { using the assumption and simplifying}

$true$

which is vacuously satisfied (i.e. by the empty code, or *skip*).

More generally, when the invariant contains reified variables, the following scheme specifies code for establishing an invariant $I(x)$ in the initial state:

assume: $hist = []$
$\wedge$ ... initialization constraints on other reified variables
achieve: $I(x)$

3.2 Specifying Disruptive Code and Deriving the Pointcut

To proceed with the inductive argument, we must maintain the invariant for all actions of the target code. Since most actions of the target code have no effect on the invariant, for efficiency it is useful to focus on those actions that might disrupt the invariant. We will then generate code for maintaining the invariant in parallel with the disruptive action. The set of all code points that might disrupt the invariant corresponds to the AspectJ concept of events that satisfy a pointcut.

An exact characterization of the disruption points is given by

$$I(x) \neq I(x'). \tag{1}$$

That is, any action that satisfies (1) as a postcondition is a disruption point. More generally, any action that satisfies a necessary condition on (1) is a potential disruption point. We can simplify (1) a little by assuming that $I(x)$ holds before the action, so all we need is to find a necessary condition on $\neg I(x')$.

In our example, we set up the following inference task:

assume: $cnt = lengthDataops \ mod \ 6$
$\wedge \ hist' = hist :: \langle \_, act, \_ \rangle$
$\wedge \ cnt' = cnt$
simplify: $\neg(cnt' = lengthDataops \ mod \ 6)$

In words, we assume that the invariant holds before an arbitrary action *act*, and that the *hist* variable is updated in parallel with *act*. Moreover, we add in a frame axiom that asserts that *act* does not change *cnt* since it is a fresh variable introduced by the invariant.

Intuitively, one would expect to derive *dataop*? as the characterization of actions that could disrupt the invariant, and that is indeed the case. Since the details of the calculation are similar to examples in [22], we omit them here, in favor of later examples that exhibit new features.

For the other invariant, we set up the following inference task:

assume: $cnt = 5 \implies data = file$
$\wedge \ cnt = lengthDataops \ mod \ 6$
$\wedge \ hist' = hist :: \langle \_, act, \_ \rangle$
$\wedge \ dataop?(act)$
$\wedge \ cnt' = (cnt + 1) \ mod \ 6$
simplify: $\neg(cnt' = 5 \implies data' = file')$

We calculate a pointcut specification as follows:

$\neg(cnt' = 5 \implies data' = file')$

$\iff$ { simplifying }

$cnt' = 5 \ \wedge \ data' \neq file'$

$\iff$ { using postcondition of dataop }

$cnt' = 5$

$\iff$ { using assumption on cnt' }

$(cnt + 1)\ mod\ 6 = 5$

$\iff$ { simplifying }

$cnt = 4$.

That is, it is only the occurrence of a *dataop* action when $cnt = 4$ that could possibly disrupt the invariant.

Generally, the task to infer a pointcut is given by the inference scheme in Figure 1.

---

assume: $I(x)$
$\wedge\ hist' = hist :: \langle \_, act, \_ \rangle$
$\wedge$ ... updates of other reified variables ...
$\wedge$ ... relevant frame conditions ...
simplify: $\neg I(x')$

Figure 1: Inference Scheme for Joinpoint Specification

The simplified result will typically contain a mixture of constraints, some of which constrain the action code (which actions might violate the invariant), and some of which constrain the state in which the action is taken.

3.3 Specification and Derivation of Maintenance Code

To complete the induction, for each potentially disruptive action (using the derived pointcut specification), we generate maintenance code to reestablish the invariant in parallel with it. Consider the second derived pointcut specification. Suppose that *act* is an action such that *dataop*?(*act*) and suppose that $cnt = 4$. In order to preserve the invariant, we need to perform a maintenance action that satisfies

assume: $(cnt = 5 \implies data = file)$
$\wedge\ cnt = lengthDataops\ mod\ 6$
$\wedge\ dataop?(act)$
$\wedge\ cnt = 4$
$\wedge\ hist' = hist :: \langle \_, act, \_ \rangle$
$\wedge\ cnt' = (cnt + 1)\ mod\ 6$
achieve: $cnt' = 5 \implies data' = file'$

The postcondition simplifies straightforwardly to the postcondition $data' = file'$ which is satisfied by an operator, say *saveData*, that saves *data* into the $file$. Similarly, we calculate the straightforward maintenance postcondition

$$cnt' = (cnt + 1)\ mod\ 6$$

for the first derived pointcut from Section 3.2.

More generally, suppose that static analysis has identified an action *act* as potentially disruptive of invariant $I(x)$. If *act* satisfies the specification

assume: $P(x)$
achieve: $Q(x, x')$

then the maintenance code can be specified as in Figure 2. In this schematic specification we compose the aspect with the base code by means of a conjunction. Note that this specification preserves the effect of *act* while additionally reestablishing the invariant I. If it is inconsistent to achieve both, then the specification is unrealizable.

---

assume: $P(x)\ \wedge\ I(x)$
$\wedge\ hist' = hist :: \langle s_0, act, s_1 \rangle$
$\wedge$ ...updates to other reified vars...
achieve: $Q(x, x')\ \wedge\ I(x')$

Figure 2: Inference Scheme for Maintenance Specification

We are not finished with this example yet. It remains to explain the mechanism whereby the parts of the induction argument, derived above, are carried out on the target system design. The next section introduces the required mechanism, and then completes the example.

4. TRANSFORMATION AUTOMATA

Program transformations have long been used to effect change on program designs, for example as in the optimizing transformations in compilers. Traditionally, a transformation has the form

$$sourcePat \rightarrow targetPat \quad \text{if } C$$

which applies to an expression *expr* in a program context if (1) *expr* matches *sourcePat* with certain bindings; i.e. $\theta = match(expr, sourcePat)$ where θ is a substitution, and (2) the condition C holds in context; i.e. $C\theta$ can be proved in context. The effect of the transformation is to replace *expr* with $targetPat\theta$.

Clearly, a transformation produces a local change in a program text. In general there is little that can be said about the semantic effect of a transformation, since *expr* can be replaced with arbitrary code. Typically however, most transformations are used to replace an expression with an equal expression (modulo context), so the effect is to preserve the semantics of the whole despite a syntactic change to a local part. We are concerned with the more general problem of whether a collection of local changes enforces a global policy and effects a global refinement.

To enforce an invariant global policy/requirement on a system, it is necessary to ensure that the invariant holds in all transitions in all system traces. We introduce the notion of a transformation automaton as the means for carrying out a systematic collection of local transformations that achieve a desired global effect.

A *transition transformation* has the form

$$[P]\{actPat\}[Q] \rightarrow [A]\{newActPat\}[B] \qquad \text{if } C$$

where P, Q, A, B are state predicates, *actPat* and *newActPat* are patterns (expressed over the specification

language and using appropriate pattern notations), and C is a state predicate that expresses the conditions of the transformation.

A transition transformation *matches* an action *act* if (1) *act* matches *actionPat* with bindings θ, and (2) *act* satisfies the pre/postcondition $[P\theta, Q\theta]$; i.e.

$$P\theta \implies wp(act, Q\theta)$$

holds, where *wp* is the weakest precondition operator [6]. The intention is that a transition transformation matches a system action either via pattern matching with the *actionPat*, or by satisfying a pre/post-condition specification, or by a combination of the two. Deciding whether an transition transformation is enabled is undecidable in general. A practical implementation of this approach must restrict the language and logic to allow efficient decision procedures. Of course, by forgoing the use of the pre/postcondition specifications, one has ordinary transformations on each transition, and therefore fast matching.

When a transition transformation matches action *act* with substitution θ, then its effect is to replace *act* by a new action

$$\text{if } C\theta \text{ then } newAct \text{ else } act$$

where *newAct* satisfies the right-hand side (RHS) specification; i.e. such that

$$\theta' = match(newAct, newActPat\theta)$$

and

$$A\theta\theta' \implies wp(newAct, B\theta\theta')$$

holds.

A *Transformation Automaton* (TA) is an automaton whose transitions are labeled with transition transformations. We write transformation automata using a Java-like syntax of the form

```
TA policyName {
    variable-declaration*
    transition-declaration*
}
```

Each variable-declaration introduces a local variable to the policy and is declared using Java-like syntax (* is used to denote zero or more occurrences).

Each transition-declaration specifies a transition transformation as described above. Policy variables can be initialized, referenced, and modified by the transition transformations. In addition, each transition transformation can have local metavariables in its patterns that are bound to system action expressions. For purposes of this paper TAs do not have a mechanism to bind system values/objects to local variables. This simplifies the presentation and process of applying TAs, but prohibits the application of more than one instance of a policy to a system design. The extensions needed to capture source system values and support multiple policy instances can be found in [23, 24].

Since TAs track both state properties and events, they can represent the checking and enforcement of a variety of kinds of requirements, ranging from event ordering to temporal logic properties and combinations of these.

Returning to the AutoSave example, we assemble a TA from the pieces of the inductive argument that were derived above:

```
TA AutoSave {
    Nat cnt
    {init} → []{}[cnt' = 0]
    {dataop?(act)}
            → [act_pre]{}[act_post ∧ cnt' = cnt + 1]
    {dataop?(act)}
            → [act_pre]{}[act_post ∧ data' = file']
                if cnt = 4
}
```

where *init* is a no-op action at the beginning of the program. The *init*-enabled transition transformation serves to initialize state and uses the postcondition $cnt' = 0$ derived in Section 3.1. The remaining transition transformations are assembled from the derived pointcut specifications (from Section 3.2) and corresponding derived maintenance code specifications (from Section 3.3). Each derived pointcut specification forms the left-hand side (LHS) and the corresponding specification of maintenance code forms the right-hand side(RHS). The predicates act_{pre} and act_{post} denote the pre- and post-conditions of *act*, respectively. Also, we use a predicate on actions in place of a pattern, here *dataop*?. This is more concise for communication purposes and avoids some of the formal noise which is necessary in particular pattern languages. Also we omit the pre- and post-conditions, action patterns, and conditions when they provide no constraints.

The AutoSave TA can be made more concise by using some abbreviations for transformation patterns that are both commonly occurring and have pleasant semantic properties. Each of these abbreviations effects a refinement - it preserves properties of the system action as well as establishing a new property.

Abbreviation	*RHS pattern*
achieve R	$[act_{pre}]\{\}[act_{post} \wedge R]$
maintain I	$[act_{pre} \wedge I]\{\}[act_{post} \wedge I]$
ensure $[P, Q]$	$[act_{pre} \wedge P]\{\}[act_{post} \wedge Q]$
ok	$\{act\}$

Figure 3: TA Abbreviations

With these abbreviations, the AutoSave TA can be expressed more compactly as

```
TA AutoSave {
  Nat cnt
  {init} → achieve [cnt' = 0]
  {dataop?(act)} → achieve [cnt' = cnt + 1 mod 6]
  {dataop?(act)} → achieve [data' = file' ]
```

if $cnt = 4$

}

or, after carrying out the straightforward syntheses,

TA AutoSave {
  Nat cnt
  $\{init\} \rightarrow \{cnt := 0\}$
  $\{dataop?(act)\} \rightarrow \{act;\ cnt := cnt + 1\ mod\ 6\}$
  $\{dataop?(act)\} \rightarrow \{act; saveData\}$ if $cnt = 4$
}

The deductive calculations that translate a logically stated policy into a TA are similar to those performed in the Finite Differencing transformation [14, 20]. They can be automated over some domains, but in general may require some user interaction.

Applying Transformation Automata

The application of a transformation automaton to a system design is accomplished by a form of sound static analysis. Requiring the analysis to be sound means that the transition transformations are applied locally to all program actions to which the transformations apply. This means that in all traces and all transitions in each trace, if a TA transition transformation applies, then it has been applied. There are no false negatives. This key property enables us to assert strong semantic claims about the design after transformation by a TA.

Since we are consumers and not developers of static analysis technology, we only informally specify the necessary techniques here. The analysis algorithms are well-known, e.g. [5, 18, 2], although practical implementations must pay careful attention to efficiency.

The strategy for applying a transformation automaton proceeds in stages, as presented below.

The first stage is a flow-sensitive interprocedural dataflow analysis that simulates the transformation automata over the Control Flow Graph (CFG) of the system design. The result of TA simulation includes (1) a map from each source control point to a representation of possible policy variable values, (2) a map from each source code action to a set of policy transitions, and (3) a summary of the state changes effected by method calls.

In the second stage, the transition transformations that label each system action are applied. Schematically, let act be a system action that is labeled with policy transition

$$act \rightarrow\ newAct \text{ if } C$$

and suppose that the control point just before act has formula V as the representation of possible policy variable values. Soundness of static analysis means that V characterizes a superset of values that the policy variables can take on over all possible system traces. As discussed above, the effect of applying the transition transformation is to replace act with

$$\text{if } C\theta \text{ then } newAct \text{ else } act.$$

Simplifying $C\theta$ with respect to V can simplify the whole conditional, especially if $C\theta$ reduces to *true* or *false*.

For example, suppose that $Transpose$ is a system action that satisfies $dataop?$. Then the AutoSave transition

$$\{dataop?(act)\} \rightarrow \{act; saveData\} \text{ if } cnt = 4$$

matches and it results in the replacement of $Transpose$ by

if $cnt = 4$
  then $(Transpose;\ saveData)$
  else $Transpose$.

If the contextual property representation V is $cnt \in \{0..5\}$, then no simplification can be performed. If V is $cnt \in \{4\}$, then the conditional simplifies to just the then-branch.

Finally, any necessary synthesis is performed on pre/post-condition specifications that have been inserted into the design. Aside from the synthesis subtasks, a TA can be applied automatically.

Returning to the AutoSave example again, the net effect of applying TA `AutoSave` to a design $D0$ resulting in design $D1$ is to enforce the invariants, allowing us to assert

$$D1 \vdash \Box\ cnt = (length \cdot dataop? \triangleright action \star hist)\ mod\ 6$$

and

$$D1 \vdash \Box\ cnt = 5 \implies data = file.$$

In this case it is also clear that D1 is a refinement of $D0$ because we have only used refinement-inducing transition transformations.

5. MORE EXAMPLES

5.1 Access Control

Essentially, access control policies prescribe which agents are allowed to access which resources. More elaborate policies may also take into account the type of access, the time of access, roles, and other features. Lampson's permission tables [11] are the basic extensional way to represent a policy – as a relation between agents/subjects/principals, and resources/objects (and possibly action-type, time, etc.). Role-Based Access Control [8] is a leading current approach to represent the permissions tables in a rule-based way that (1) is natural and compact and (2) allows easier maintenance/evolution than a tabular/relational format.

Our overarching concern is to formally specify and enforce cross-cutting requirements on a system. Many requirements can be specified as state invariants that are given by a state predicate that is required to hold before and after each system action. Other requirements place constraints on the order of system actions. Access control policies are requirements on both events (an action to access a protected resource) and state properties (the current permission table).

Intuitively, access control (or authorization) is a requirement that whenever a system action act whose principal or agent a accesses resource r, then a has current permission to access r. Although the policy is easy to state in a positive

way (i.e. what behaviors are allowed), it is applied with the understanding that the target design may not satisfy the requirement, and there may be a need to deal with exceptions to it. If we think of the policy as expressing normal behavior, then ultimately the specification of it must deal with possible departures from normal behavior.

In order to formalize access control we need some way to discuss the principal of an action, and current permissions. Both of these are extra-computational entities, so we must reify them in order to mention them in a formal requirement.

Reifying Agency and Permissions

To formalize access control, we must reify the agents who are the initiators of system actions. To do so, we introduce a finite type *Agent*, and label each action in a trace with an *Agent*. Not all labelings make sense - there are constraints on consistent labelings. The main constraint is that if system action α_i is labeled with agent a (meaning that a is the principal behind action α_i), and the control of the system naturally flows from α_i to α_{i+1}, then α_{i+1} is also labeled with agent a. Since each action in a trace has a unique label, we write $prin(act)$ to denote that label.

The semantics of the system is now all possible system traces with all possible consistent labelings of system actions with agents.

The other reification we need is the permissions. We add a finite type *Resource* and a finite map $ACP : Agent \times Resource \rightarrow Boolean$ (Access Control Permissions). We'll assume that ACP is a variable and that it can change from system state to system state. It is not obvious what kinds of constraints to put on these changes, so we won't assume any.

The semantics of the system is now all possible system traces with all possible consistent labelings of system actions with *Agent*s and with all possible values of ACP at system states.

Comments on this semantics:

1. *Messages* – A service (method) that passively waits for control and data, acquires the agency of the invoker. On the other hand, a process B that receives a message from process A naturally continues on its course with its agency unchanged.

2. *Delegation* – The situation in which agent A temporarily endows agent B with some of A's permissions is handled in traces by B temporarily gaining additional permissions. What is not modeled is the situation in which A passes her credentials to B so that B can act as A – credentials are an implementation concept used to satisfy requirements on authentication and access control. The semantic model here is more abstract and admits both credential-based implementations as well as others.

3. *Permission table modifications* – Agent actions that modify the ACP (permission table) are not modeled. Again, the idea is to abstract away implementation detail. A more elaborate semantical model would include (i) the principal behind the actions that change the access control policy (ACP) and (ii) permissions to effect such changes.

This is a fairly simple semantics for reifying identity in a system. Doubtless a more elaborate model could be constructed. This one is accurate enough for present purposes.

Access Control Requirement

We can now specify the access control requirement on system S; for each trace $tr : Traces(S)$, transition $\langle s, act, s' \rangle \in tr$, and resource $r : Resource$:

$$access?(act, r) \implies s.ACP(principal(act), r)$$

where $access?(act, r)$ holds if action act directly accesses resource r. The requirement states that if the current action directly accesses resource r, and the principal behind the action is a, then $ACP(a, r)$ holds in the prestate (i.e. agent a has permission to access resource r). Naturally, there are many variants and elaborations of this requirement, but this form lets us treat the essential ideas.

The reader should not confuse a simple clear specification with the ease of implementing it - accurate tracking of identity in a system is a notoriously difficult problem.

Enforcing the Requirement

In order to correctly realize the requirement in the target code, we proceed by direct synthesis.

First, we can derive a joinpoint specification as a necessary condition that a system action violates the requirement. We instantiate the inference scheme in Figure 1 as follows.

assume: $hist' = hist :: \langle s, act, s' \rangle$
simplify: $\neg(access?(act, r) \Rightarrow s.ACP(prin(act), r))$

and calculate a pointcut specification as follows:

$$\neg(access?(act, r) \implies s.ACP(prin(act), r))$$

$$\iff \quad \{ \text{ simplifying } \}$$

$$access?(act, r) \ \wedge \ \neg s.ACP(prin(act), r)$$

as one expects. The constraint on the system action, $access?(act, r)$ will serve as a joinpoint specification, and the state predicate $\neg s.ACP(prin(act), r)$ will serve as the condition on a policy transition in a TA.

Next, suppose that the current action act satisfies the joinpoint specification and has the particular specification

assume : $P(x)$
achieve : $Q(x, x')$

then the code to enforce the requirement can be specified by instantiating the inference scheme from Figure 2.

assume: $P(x) \wedge access?(act, r)$
$\wedge\ hist' = hist :: \langle s, act, s' \rangle$
achieve: $Q(x, x')$
$\wedge\ (access?(act, r) \Rightarrow s.ACP(prin(act), r))$

We refine the postcondition as follows:

$Q(x, x') \wedge (access?(act, r) \Rightarrow s.ACP(prin(act), r))$

$\Longleftrightarrow$ { simplifying }

$Q(x, x') \wedge s.ACP(prin(act), r)$

$\Longleftrightarrow$ { ordering the evaluation }

if $s.ACP(prin(act), r)$
then $Q(x, x')$
else false.

We have derived a postcondition specification for an action that would jointly realize both the current action and satisfy the access control requirement. The fact that the postcondition is *false* in one case is the essence of the semantic problem - we have deduced inconsistency between the current system design and the access control policy. Although we have calculated a correct refinement, it specifies a step in the design that is unimplementable! Just as any specification refines to the inconsistent specification, it is mathematically sound to have an action specification refine to an inconsistent action specification.

A transformation automaton to realize the policy as calculated above is

TA AccessControl {
Resource r
$\{access?(act, r)\} \rightarrow$ achieve $[false]$
if $\neg ACP(prin(act), r)$
}

What we can say is that if the source design $D0$ satisfies invariant R; i.e. $D0 \vdash \Box R$, and $D1$ results from the application of TA AccessControl, then

$$D1 \vdash (R \wedge AC)\ \mathcal{W}\ false$$

where AC is the access control invariant and $\mathcal{W}$ is the *unless* modality of temporal logic [12]. In words, $D1$ traces satisfy the state/transition property $R \wedge AC$ up to the point (if any) that the transition has a *false* postcondition. Another way of putting it is that the composition of the policy and $D0$ has preserved its safety properties, but has possibly decreased its liveness properties.

Of course, we cannot have a program with an unimplementable action in it. The solution is to weaken the postcondition to something implementable. The following TA specifies the throwing of an exception

TA AccessControl1 {
Resource r
$\{access?(act, r)\} \rightarrow$ {throw new error("...")}
if $\neg ACP(prin(act), r)$
}

Current work on self-healing systems attempts to deal with situations like this, albeit dynamically. Ideally, there is a way to lift out of the black hole of an inconsistency and take an action that allows the system to continue toward its goals.

5.2 A Simple Information Flow Policy

Consider the following simple security policy which is adapted from [19]. If a process ever reads from a particular file f, it is henceforth not allowed to send any messages. The policy states an information flow requirement. One might want to automatically enforce an instance of this policy on an applet downloaded onto a personal computer.

This example is distinguished from previous examples in that it is a pure event ordering constraint, whereas AutoSave is a state invariant and AccessControl constrains an action and the state in which it executes.

The reification of history allows us to represent the event ordering as a state invariant. If *Send* matches any *send*(..) action and *Readf* matches any action that reads file f, then the policy can be expressed as an invariant: for all traces $tr : Traces(S)$ and transitions $\langle s, act, s' \rangle \in tr$

$$Send(act) \Rightarrow \neg\exists(a)(a \in action \star hist\ \wedge\ Readf(a))$$

however, this seems less than straightforward. Constraints on the order of events are often more naturally expressed using the tools of language theory: regular expressions, recognition automata, grammars. Using regular expressions for example allows the straightforward formulation

$$Send^* Readf^*$$

and a corresponding automaton is similarly clear. Note that all of these formulations specify normal or allowed behaviors but do not prescribe what to do with violations.

Our approach is to generate a TA that effects the specified policy, allowing developers to fill in how to handle violations.

TA InfoFlow {
Boolean *rf*
$\{init\} \rightarrow$ achieve $[rf' = false]$
$\{Send\} \rightarrow$ ok if $\neg rf$
$\{Readf\} \rightarrow$ achieve $[rf' = true]$
}

where *rf* flags whether a *Readf* action has occurred. Using a derivation similar to that for AutoSave and AccessControl, we can derive the point of inconsistency (sending when condition *rf* holds). Here we manually weaken the inconsistent specification to *abort* resulting in

TA InfoFlow {
    Boolean rf
    $\{init\} \rightarrow$ achieve $[rf' = false]$
    $\{Send\} \rightarrow$ abort if rf
    $\{Readf\} \rightarrow$ achieve $[rf' = true]$
}

6. RELATED WORK

This work ties together research in a wide range of topic areas. The refinement view offers the opportunity to abstract aspects to the level of requirement specification and to treat aspect weaving as a powerful new tool for generating specification refinements. There is an opportunity to unify aspect weaving with other related techniques, including intrusion detection [26], Software Fault Isolation [7], security policy enforcement [19], and others, in addition to software development by refinement.

Runtime verification is a recent field that foregoes full program verification in favor of runtime monitoring of code with respect to a specified property of interest [3, 9]. Transformation automata can be seen as a generalization of runtime verification. Although we haven't emphasized it, when static analysis cannot decide whether a policy transition applies, then the decision must be pushed to runtime when more information is available. As such, runtime monitoring of a property then becomes a special case of applying a TA in which we defer all decisions to runtime. The static analysis performed in our approach has the effect of optimizing the runtime monitors - if we can prove statically that a certain property holds at a code location for all behaviors, then there is no need to monitor it. Also, static analysis may be able to simplify the monitoring code without eliminating it entirely, resulting in lower overhead.

The next step is to both monitor the code and take action when the policy is about to be violated. Schneider [19] defines a class of enforceable security policies as a subclass of safety properties, and uses a form of finite state machine (labeled with an event vocabulary) to express them. The effect of applying a security policy is to abort the system whenever it is about to violate the policy. In [7] the authors inject the policy automaton at each code location and then use partial evaluation to optimize away all or most of the inlined code. In [4] Colcombet and Fradet propose a similar approach except that static analysis (vs partial evaluation) is used to optimize away unnecessary runtime code. Static analysis can exploit more context and can in general optimize away more of the runtime monitoring code.

TA's generalize previous work in transformations in the following ways. The automaton provides behavioral context for the transition transformations, thereby providing more flexible and coordinated control over when they are applied. By packaging a collection of related transformations and using static analysis to explore the behaviors of the target system code, a new range of global effects are enabled. Also, the use of optional pre/post-condition specifications for both matching and target code generation is unique to our knowledge, although our Refine [17] system allowed a limited postcondition capability in specifying target code.

Recent work in AOSD has extended AspectJ concepts in various dimensions. Several authors have proposed generalizing pointcuts to take behavioral context into account, e.g. tracecuts [1], Jasco [25], PQL [13], [27], and our own policy automata [23]. Other works have increased the amount of static and reflective context that can be picked up at program points. TA's generalize previous work on AOP, including work on behavioral pointcut specifications. One can include stacks and other data structures internally to gain full computability power. Also, one could write TA's that have arbitrarily complex pre/post-conditions on their RHS which would entail arbitrarily hard synthesis problems to effect them. Effective implementations of TA's would likely place restrictions on the expressiveness to gain full automation of the enforcement process. As a special case, if no synthesis tasks appear on the RHS of transitions, then application of a TA can be fully automatic.

To our knowledge, the work on retrenchment by Poppleton and Banach [16] is the only other work that confronts the issue of transformations that impose limitations on a design from a refinement point-of-view. Their solution is to define a generalization of refinement that allows preconditions to strengthen and postconditions to weaken in some situations. Their approach is broadly consistent with the discussion above: enforcing a policy typically strengthens the guards on actions, and in the case of a derived inconsistency, we are forced to weaken an inconsistent postcondition to make progress.

7. CONCLUDING REMARKS

This paper takes a step in the direction of integrating and cross-fertilizing the two fields of AOSD and software development by refinement. The unification requires generalizations of concepts from both fields.

This paper advocates the following process for enforcing global system requirements. First, a natural specification of a requirement is translated, via some deductive calculation, into a transformation automaton. Static analysis simulates the TA over the target system design, and then applies the component transformations of the TA. The resulting transformed design satisfies the given requirement, and under certain conditions, is a refinement of the starting design. The composition process preserves the invariants of the starting design, but may reduce its liveness. That is, the enforcement of safety and security policies on a design may result in the curtailment of some behaviors that violate the policies.

This overall process enriches previous approachs to refinement by offering an automated technique for folding requirements into a design. The refinement process starts by focusing on a subset of requirements, say, to meet key functional and performance needs. Then, one can add in other requirements incrementally. Feedback from the enforcement process informs the revision of earlier design decisions, hopefully leading to designs that satisfy all requirements under a broader range of conditions.

Acknowledgements: This work benefited from discussions with Klaus Havelund, Dusko Pavlovic, Peter Pepper, and from comments from the reviewers. This work was partially supported by the Office of Naval Research under Grant N00014-04-1-0727 and by the US Department of Defense.

8. REFERENCES

[1] Allan, C., Avgustinov, P., Christensen, A. S., Hendren, L. J., Kuzins, S., Lhotk, O., de Moor, O., Sereni, D., Sittampalam, G., and Tibble, J. Adding trace matching with free variables to AspectJ. In *Proceedings of OOPSLA* (2005), pp. 345–3640.

[2] Brat, G., and Venet, A. Precise and scalable static program analysis of NASA flight software. In *Proceedings of the 2005 IEEE Aerospace Conference* (2005).

[3] Cohen, D., Feather, M. S., Narayanaswamy, K., and Fickas, S. S. Automatic monitoring of software requirements. In *ICSE '97: Proceedings of the 19th international conference on Software engineering* (1997), ACM Press, pp. 602–603.

[4] Colcombet, T., and Fradet, P. Enforcing trace properties by program transformation. In *Proc. 27th ACM Symp. on Principles of Programming Languages* (Jan. 2000), pp. 54–66.

[5] Cousot, P., and Cousot, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1977), ACM, pp. 238–252.

[6] Dijkstra, E. W. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1976.

[7] Erlingsson, U., and Schneider, F. SASI enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigms Workshop* (Ontario, Canada, September 1999).

[8] Ferraiolo, D., and Kuhn, D. Role based access control. In *15th National Computer Security Conference* (1992).

[9] Havelund, K., and Rosu, G. Monitoring Java programs with Java PathExplorer. In *Electronic Notes in Theoretical Computer Science* (2001), K. Havelund and G. Rosu, Eds., vol. 55, Elsevier.

[10] Kiczales, G., and et al. An Overview of AspectJ. In *Proc. ECOOP, LNCS 2072, Springer-Verlag* (2001), pp. 327–353.

[11] Lampson, B. W. Protection and access control in operating systems. *Operating Systems, Infotech State of the Art Report 14* (1972), 309–326.

[12] Manna, Z., and Pnueli, A. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, New York, 1992.

[13] Martinand, M., Livshits, B., and Lam, M. Finding application errors and security flaws using pql: a program query language. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object Oriented Programming Systems Languages and Applications* (2005), ACM Press.

[14] Paige, R., and Koenig, S. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems 4*, 3 (July 1982), 402–454.

[15] Pavlovic, D., and Smith, D. R. Composition and refinement of behavioral specifications. In *Proceedings of Sixteenth International Conference on Automated Software Engineering* (2001), IEEE Computer Society Press, pp. 157–165.

[16] Poppleton, M., and Banach, R. Retrenchment: Extending the reach of refinement. In *Proceedings of the Fourteenth Automated Software Engineering Conference* (1999), IEEE Computer Society Press, pp. 158–165.

[17] Reasoning Systems, Palo Alto, CA. *The $REFINE^{TM}$ User's Guide*, 1985.

[18] Reps, T., Horwitz, S., and Sagiv, M. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of the Twenty-Second ACM Symposium on Principles of Programming Languages* (1995), ACM, pp. 49–61.

[19] Schneider, F. Enforceable security policies. *ACM Transactions on Information and System Security 3*, 1 (February 2000), 30–50.

[20] Smith, D. R. KIDS – a semi-automatic program development system. *IEEE Transactions on Software Engineering Special Issue on Formal Methods in Software Engineering 16*, 9 (1990), 1024–1043.

[21] Smith, D. R. Mechanizing the development of software. In *Calculational System Design, Proceedings of the NATO Advanced Study Institute*, M. Broy and R. Steinbrueggen, Eds. IOS Press, Amsterdam, 1999, pp. 251–292.

[22] Smith, D. R. Aspects as invariants. In *Automatic Program Development: a Tribute to Robert Paige* (2006), O. Danvy, F. Henglein, H. Mairson, and A. Pettorosi, Eds., Springer-Verlag LNCS. (earlier version in Proceedings of GPCE-04, LNCS 3286, 39-54).

[23] Smith, D. R., and Havelund, K. Automatic enforcement of error-handling policies. Tech. rep., Kestrel Technology, September 2004.

[24] Smith, D. R., and Havelund, K. Enforcing safety and security policies. Tech. rep., Kestrel Technology, December 2005.

[25] Vanderperren, W., Suvee, D., Cibran, M., and de Fraine, B. Stateful aspects in JAsCo. In *Proceedings of SC 2005* (2005), Springer-Verlag LNCS.

[26] Vigna, G., and Kemmerer, R. NetSTAT: A Network-based Intrusion Detection System. *Journal of Computer Security 7*, 1 (1999), 37–71.

[27] Walker, R., and Viggers, K. Implementing protocols via declarative event patterns. In *SIGSOFT Foundations of Software Engineering (FSE04)* (2004), ACM Press, pp. 159–169.

Typing For a Minimal Aspect Language

Preliminary Report

Peter Hui James Riely
DePaul University
{phui,jriely}@cs.depaul.edu

Abstract

We present a preliminary report on typing systems for polyadic μABC, aspect oriented programming—pointcuts and advice—and nothing else. Tuples of uninterpreted names are used to trigger advice. The resulting language is remarkably unstructured: the least common denominator of the pi-calculus and Linda. As such, developing meaningful type systems is a substantial challenge.

Our work is guided by the translation of richly typed languages into μABC, specifically function- and class-based languages augmented with advice. The "impedance mismatch" between source and target is severe, and this leads us to a novel treatment of types in μABC.

1. Introduction

Research on the foundations of aspect-orientation has followed several directions. Much work has found inspiration in functional languages (for example [4, 9]), whereas others have looked to objects (for example [3]). These works follow the view that aspects transform code from some underlying paradigm. In line with our prior work, this paper follows a different route, attempting to understand aspects, in so far as possible, in isolation.

By removing the underlying computational mechanisms, however, it is not clear what aspects are meant to advise. In μABC, which we study here, aspects advise tuples of names, drawing inspiration from the pi calculus and coordination languages, such as Linda. A surprisingly expressive computational model emerges, but it is not without difficulties. In particular, the language is almost shockingly unstructured, making any form of analysis seem rather hopeless. Here we make a first attempt at redressing this situation.

μABC was introduced in [2] to study as aspects "as primitive computational entities on par with objects, functions and horn-clauses". In that paper, we sketched an encoding of core minAML [11, 9] into μABC, but did not provide a full translation. Indeed, we observed that

> μABC was deliberately designed to be a small calculus that embodies the essential features of aspects. However, this criterion makes μABC an inconvenient candidate to serve in the role of a meta-language that is the target of translations from "full-scale" aspect languages.

Sixth International Workshop on Foundations of Aspect-Oriented Languages (FOAL 2007), March 13, 2007, Vancouver, BC, Canada.

Echoing this sentiment, Ligatti, Walker and Zdancewic [9] note that "it is unclear what sort of type theory would be needed to establish that [...] translations [into μABC] are type-preserving." Here, we take the first steps at providing such a theory, in an attempt to bridge the gap between the chaos of pure aspects and the relatively well-behaved worlds of functions and objects.

As our point of departure, we use the polyadic version of μABC introduced in [5]. In this variant, the events upon which advice triggers are otherwise uninterpreted tuples of ordered names. When modeling a functional language, the elements of the tuple may represent the function and its arguments. When modeling an object language, the elements instead may represent the source and target of a message, along with the message name and arguments. μABC itself imposes no interpretation.

When defining a pointcut in μABC, one must specify the arity of the events (ie, the length of the tuples) upon which it will trigger. One may treat all of the elements of the tuple as arguments—in which case the pointcut will trigger on any event of that arity—or one may fix some names in the pointcut so that the pointcut will only trigger on events that include the same name in the corresponding position. In addition, one may specify bounded matching, so that any name ordered below the one specified will serve to satisfy the pointcut. In this way μABC advice resembles tuple matching Linda.

The chief insight of our work is that the ordering on names suffices to encode simple type systems for function and object languages, as long as one may impose some structure on names. By systematically selecting bounds, and relating the bounds between elements of a tuple, one may specify constraints on the tuple shapes which are allowable. For example, one might require that if the first element of a triple is a subname of $\mathsf{int}\to\mathsf{int}$, then the second must be a subname of int, and the third a subname of int^{-1}. Here int, $\mathsf{int}\to\mathsf{int}$ and int^{-1} are simply names, albeit with some structure. One may view this as a protocol that imposes an interpretation on subnames of int, $\mathsf{int}\to\mathsf{int}$ and int^{-1}. Tuples that use such subnames must satisfy requirements such as that stated above.

One would expect that the protocol used by a functional language would be different than that of an object language, and again from a logic language. μABC may be adapted to any system by specifying both a structure on certain names (ie, those that correspond to types) and a protocol for tuples that "match" them.

We are interested in discovering a general theory of such protocols and establishing its validity in μABC. As of yet, we have not reached this ideal, but we have discovered some intermediate results that may be of interest to the FOAL community.

As a first step toward full typing, we have specified a *sorting* for μABC, in the flavor of sorting systems for the pi calculus. The sorting is sufficient to guarantee that computation never terminates. We believe that the sorting system is an important first step toward developing a proper typing system; however, in terms of semantic

equivalence, sorting itself is clearly inadequate: all well-sorted programs are indistinguishable! Being a continuation calculus, like pi, there is no clear notion of *value* at which one might terminate. Our vision of typing is that well-typed terms either nonterminate or terminate on a tuple of a particular shape, allowing a richer notion of equivalence based on barbed congruence [10]. However, we have not yet developed the technical machinery to specify this.

Further, we have defined translations from function and class based languages into well-sorted μABC and, to a certain extent, established their correctness. We hesitate to say that we have *proved* the correctness of these because the results are stated with respect to a "structural congruence". This is a problem for two reasons.

First, as noted above, we have not yet developed techniques to specify that any structural congruence is "reasonable". To establish this requires a meaningful notion of semantic equivalence.

Second, the structural congruence presented here is defined, in part, using the translation itself. This is clearly inadequate. As we discuss in Section 4, the standard semantics of aspect-functional languages [9, 8] specify that proceed substitutions are performed early, at the time of advice lookup; whereas μABC performs them late, on demand. This creates technical difficulties relating the two languages, which we believe are best solved by slowing down proceed substitutions in the function language, for example by using explicit substitutions [1]. We do not do this here, however, and therefore introduce a questionable structural rule to solve the problem.

The remainder of the paper proceeds as follows. In the next section we review the syntax and operational semantics of μABC. Section 3 presents the sorting system. In Section 4, we present a λ-calculus with advice and its translation into μABC. Given the caveats stated above, we demonstrate the correctness of the translation. The following section does the same for a small object language.

2. μABC

In this section we present the syntax and evaluation semantics of polyadic μABC. We give some examples of evaluation here; further examples can be found in Section 4.3.

We assume disjoint sets of *names*, ranged over by *a–y* and *proceed names*, ranged over by z. Names include int, self, Object as well as the structured names used in Sections 4 and 5. Integers and integer-valued expressions are also names, each a proper subname of int. This treatment of integer-valued expressions simplifies examples. The syntax of μABC is as follows.

SYNTAX

$U,V,S,T ::= \cdot \mid U,\hat{a} \mid U,a$	Events
$P,Q ::= \cdot \mid P,\hat{a} \mid P,x\!:\!s$	Pointcut Atoms
$A,B ::= \{z.P \to M\}$	Advice
$D,E ::= \cdot \mid D\,;\, \mathsf{new}\, a\!:\!s \mid D\,;\, \mathsf{adv}\, A$	Declarations
$M,N ::= \mathsf{call}\langle U\rangle \mid z\langle U\rangle \mid \vec{A}\langle U\rangle \mid D\,;\, M$	Terms

An *event*, U, is sequence event atoms, where "$\cdot$" is the empty sequence. *Event atoms* are either *exact* names, decorated with a circumflex, or *inexact* names, which have no decoration.

A *pointcut*, P, is a sequence of pointcut atoms. *Pointcut atoms* are either exact or inexact. The exact pointcut atom $\hat{a}$ matches only the exact event atom $\hat{a}$. The inexact pointcut atom $x\!:\!s$ matches any inexact event atoms whose name is a proper subname of name s. At runtime x is bound to the matching name; for this reason we call it a *pointcut variable*.

Advice has the form $\{z.P \to M\}$, where z is a *proceed variable*, P is a pointcut, and M is the body of the advice. The proceed variable z and the pointcut variables in P are bound in M. We elide the proceed variable when it does not occur free in the advice body, writing simply $\{P \to M\}$.

A *declaration sequence*, D, is a sequence of declarations. The declaration $\mathsf{new}\, a\!:\!s$ declares name a as a fresh subname of s (s may also be fresh, in which case a may only be matched exactly). The declaration $\mathsf{adv}\, A$ declares advice A.

A *term* M may be prefixed with a declaration sequence $D\,;\, M$. The new names declared in D are bound in M. We identify $(D\,;\, E)\,;\, M$ with $D\,;\, (E\,;\, M)$ and $\cdot\,;\, M$ with M.

Each term has exactly one "current" event. A term consists of a sequence of declarations, followed by the current event. Events are marked either with a call, a proceed variable, or an advice sequence. In $\vec{A}\langle U\rangle$, U is the current event and $\vec{A}$ is a sequence of pending advice. Advice is executed right-to-left.

Example 1. Consider the advice

$$\{z.\underbrace{\hat{f},x\!:\!\mathsf{int},y\!:\!\mathsf{int}}_{\text{Pointcut}} \to \underbrace{z\langle \hat{f},y,x\rangle}_{\text{Body}}\}.$$

This advice is triggered when the current event is a triple whose first atom is exactly $\hat{f}$, and whose second and third atoms are both inexact atoms whose names are subnames of int. When it executes, it switches its second and third names, and proceeds on the new event. Supposing that this is the most recently declared advice, the term

$$\underbrace{\mathsf{call}\langle \hat{f},10,20\rangle}_{\text{Current Event}}$$

evaluates to

$$\vec{B},\{z.\hat{f},x\!:\!\mathsf{int},y\!:\!\mathsf{int} \to z\langle \hat{f},y,x\rangle\}\langle \hat{f},10,20\rangle$$

where $\vec{B}$ is previously declared advice that triggers on the same event. At this point the term evaluates to

$$\vec{B}\langle \hat{f},20,10\rangle. \qquad \square$$

Example 2. Consider the following declaration:

$$D \triangleq \mathsf{adv}\, A\,;\, \mathsf{adv}\, B$$
$$A \triangleq \{z.\hat{f},x\!:\!\mathsf{int} \to M\}$$
$$B \triangleq \{z.\hat{f},x\!:\!\mathsf{int} \to z\langle \hat{f},42\rangle\}$$

D declares two pieces of advice. Both triggered when the first name of the current event is exactly $\hat{f}$, and the second name is a subname of int.

Evaluation of $D\,;\, \mathsf{call}\langle \hat{f},10\rangle$ proceeds as follows. The call triggers advice lookup. Since both pieces of advice match the current event, they are both enqueued. B executes first.

$$D\,;\, \mathsf{call}\langle \hat{f},10\rangle \to D\,;\, A,B\langle \hat{f},10\rangle$$

B changes the second atom of the event to 42, and proceeds on the next advice

$$\to D\,;\, A\langle \hat{f},42\rangle$$

and then the body of A is executed.

$$\to D\,;\, M[x := 42] \qquad \square$$

We now present the operational semantics of the language. Declarations, D, are used in several definitions. We disallow alpha conversion on the new names in a declaration when the declaration is treated as independent syntax—while a and b are bound in the term "$\mathsf{new}\, a\,;\, \mathsf{new}\, b\,;\, \mathsf{call}\langle a,b\rangle$," they are free in the declaration sequence "$\mathsf{new}\, a\,;\, \mathsf{new}\, b$."

We begin by defining some auxiliary relations. We write $D \triangleright a\!:\!s$ to indicate that a is properly below s in the order defined by D. Thus

if $D = \mathsf{new}\ s\!:\!t\,;\ \mathsf{new}\ a\!:\!s$ then $D \triangleright a\!:\!t$ and $D \triangleright a\!:\!s$. The relation is irreflexive; thus $D \triangleright a\!:\!a$ never holds.

The *match* relation takes a pointcut P, an event U, and returns an appropriate *substitution* σ, if one exists. A substitution is a finite mapping on names. For instance, if D contains $\mathsf{new}\ a\!:\!\mathsf{int}$, then $D \triangleright match(x\!:\!\mathsf{int}, \hat{b})(a, \hat{b}) = (a := x)$. If D contained $\mathsf{new}\ a\!:\!\mathsf{bool}$ instead, then $D \triangleright match(x\!:\!\mathsf{int}, \hat{b})(a, \hat{b})$ would be undefined. Matching is sensitive to exactness; thus $D \triangleright match(x\!:\!\mathsf{int}, \hat{b})(a, b)$ is always undefined, since b is exact in the pattern but inexact in the event.

AUXILIARY RELATIONS $(D \triangleright a\!:\!s)$ $(D \triangleright match(P)(U) = \sigma)$

$\sigma ::= \cdot \mid \sigma, x := a$ Name Substitutions

$D \triangleright a\!:\!s$ if $D \ni \mathsf{new}\ a\!:\!s$

$D \triangleright a\!:\!s$ if $D \triangleright t\!:\!s$ and $D \ni \mathsf{new}\ a\!:\!t$

$D \triangleright match(\cdot)(\cdot) = \cdot$

$D \triangleright match(P, \hat{a})(U, \hat{a}) = \sigma$ if $D \triangleright match(P)(U) = \sigma$

$D \triangleright match(P, x\!:\!s)(U, a) = (\sigma, x := a)$ if $D \triangleright match(P)(U) = \sigma$ and $D \triangleright a\!:\!s$ and $x \notin dom(\sigma)$

Write "$D \triangleright match(P)(U)$" if $D \triangleright match(P)(U) = \sigma$ for some σ. Write "$D \triangleright match(A)(U)$" if $A = \{\_.P \to \_\}$ and $D \triangleright match(P)(U)$.

There are two evaluation rules, shown below:

EVALUATION $(M \to N)$

$D\,;\, \mathsf{call}\langle U \rangle \to D\,;\, \vec{A}\langle U \rangle$ if $\vec{A} = \left(A \mid D \triangleright match(A)(U) \text{ and } A \in D\right)$

$D\,;\, (\vec{A}, \{z.P \to M\})\langle U \rangle \to D\,;\, M[z := \vec{A}, \sigma]$ if $D \triangleright match(P)(U) = \sigma$

The first rule states that call is executed by searching D for any advice triggered by the current event. All matching advice then advises the event. The second rule states that if there is advice advising the current event, then the current state is replaced by the body of the advice, with the appropriate substitutions. If the advice list is empty, or if the pointcut P does not match the event U, then evaluation is *stuck*. The sorting system of the next section rules out stuck terms.

3. Sorting

To avoid getting stuck, two invariants must be preserved by evaluation:

1. if a piece of advice proceeds, there must be at least one piece of advice in the advice queue, and
2. if a piece of advice proceeds and it modifies the current event, it must be certain to do so in such a way that it still satisfies the pointcuts of any remaining advice in the advice queue.

Example 3. In the absence of any other advice declarations, the following μABC program violates condition (1) above:

$$\mathsf{adv}\,\{z.\hat{f}, \hat{g} \to z\langle \hat{f}, \hat{g} \rangle\}\,;\, \mathsf{call}\langle \hat{f}, \hat{g} \rangle$$

The term evaluates to $\cdot\langle \hat{f}, \hat{g} \rangle$, which is stuck. Since there is no additional advice, the use of the proceed variable in the advice is malformed. □

Example 4. The following μABC program violates condition (2) above:

$$\begin{array}{l} \mathsf{adv}\,\{z.\hat{f}, x\!:\!\mathsf{int} \to z\langle \hat{f}, 42 \rangle\}\,; \\ \mathsf{adv}\,\{z.\hat{f}, x\!:\!\mathsf{int} \to \mathsf{new}\ g\,;\, z\langle \hat{g} \rangle\}\,; \\ \mathsf{call}\langle \hat{f}, 10 \rangle. \end{array}$$

In this example, the current event is $\langle \hat{f}, 10 \rangle$, and after evaluation, there will be two pieces of advice queued up advising it. Since advice is queued in LIFO order, the second piece of advice will trigger first. It declares a new name g, changes the current event to the tuple $\langle \hat{g} \rangle$, and proceeds on the new event. The result is that the first piece of advice now advises the new event $\langle \hat{g} \rangle$, but its pointcut is no longer satisfied by the current event. □

In this section, we present a sorting system that guarantees progress and preservation.

The sort of an event is itself is given using the same syntax as events. To distinguish the two uses, we use S, T for sorts and U, V for events.

The sort of an event computes the bounds on inexact atoms in the expected way. For instance, if we have declared $\mathsf{new}\ a\!:\!s$ and $\mathsf{new}\ b\!:\!t$, then the event $\langle a, b, \hat{c} \rangle$ has sort $\langle s, t, \hat{c} \rangle$.

We sort advice based on its pointcut. For instance, the advice $\{f, x\!:\!a, y\!:\!b \to M\}$ has sort $\langle f, a, b \rangle$.

If an advice uses its proceed variable, we say that it is *nonfinal*; if it does not use its proceed variable, we say that it is *final*. Nonfinal advice of sort S also has sort S final. For example, $\{z.\hat{f}, x\!:\!\mathsf{int}, y\!:\!\mathsf{int} \to \mathsf{call}\langle \hat{g}, y, x \rangle\}$ can be given sorts $\langle \hat{f}, \mathsf{int}, \mathsf{int} \rangle$ and $\langle \hat{f}, \mathsf{int}, \mathsf{int} \rangle$ finalized; it ignores its proceed variable and hence any advice declared before. The advice $\{z.\hat{f}, x\!:\!\mathsf{int}, y\!:\!\mathsf{int} \to z\langle \hat{g}, y, x \rangle\}$, instead, can be given only sort $\langle \hat{f}, \mathsf{int}, \mathsf{int} \rangle$; this advice is *nonfinal*. If advice of sort S final has been declared, we say that sort S has been *finalized*.

We sort with respect to the *environment*, Γ, that keeps records the sorts of names and advice, as well as the sorts that have been finalized. These concepts are formalized below:

ENVIRONMENTS

$\Gamma, \Delta ::= \cdot \mid \Gamma, (a\!:\!s) \mid \Gamma, (z\!:\!S) \mid \Gamma, (S \text{ finalized})$

We require that all environments be well formed, in the sense that each name a occur at most once on the lefthand side of a declaration $a\!:\!s$. Formally, the environment "Γ, Δ" consisting of the union of Γ and Δ is undefined if any name occurs in the domain of both Γ and Δ.

The sorting rules for pointcuts and events are as follows. Pointcuts produce an environment Δ which includes all the names bound by the pointcut.

SORTING $(\vdash P : S \triangleright \Delta)$ $(\Gamma \vdash U : S)$

$\vdash \quad (\cdot) : (\cdot) \triangleright (\cdot)$

$\vdash \quad (P, \hat{a}) : (T, \hat{a}) \triangleright (\Delta)$ if $\vdash P : T \triangleright \Delta$

$\vdash (P, x\!:\!s) : (T, s) \triangleright (\Delta, x\!:\!s)$ if $\vdash P : T \triangleright \Delta$

$\Gamma \vdash (\cdot) : (\cdot)$

$\Gamma \vdash (U, \hat{a}) : (S, \hat{a})$ if $\Gamma \vdash U : S$

$\Gamma \vdash (U, a) : (S, s)$ if $\Gamma \vdash U : S$ and $\Gamma \ni a\!:\!s$

$\Gamma \vdash (U, a) : (S, s)$ if $\Gamma \vdash U : S$ and $\Gamma \ni a\!:\!t$ and $\Gamma \vdash t : s$

Similarly to pointcuts, the sorting of declarations extracts the sorts of the names declared in D, as well as the sorts have been finalized as a result of the advice declared in D. The judgement takes the form $\Gamma \vdash D \triangleright \Delta$.

SORTING $(\Gamma \vdash A : S)$ $(\Gamma \vdash A : S \text{ final})$ $(\Gamma \vdash D \triangleright \Delta)$ $(\Gamma \vdash M \text{ ok})$

$\Gamma \vdash \{z.P \to M\} : S$ if $\vdash P : S \triangleright \Delta$ and $\Gamma, z\!:\!S, \Delta \vdash M$ ok

$\Gamma \vdash \{z.P \to M\} : S$ final if $\vdash P : S \triangleright \Delta$ and $\Gamma, \Delta \vdash M$ ok

$\Gamma \vdash \quad \cdot \triangleright \cdot$

$\Gamma \vdash D\,;\, \mathsf{new}\ a\!:\!s \triangleright \Delta, a\!:\!s$ if $\Gamma \vdash D \triangleright \Delta$ and $a \notin \Gamma, \Delta$

$\Gamma \vdash \quad D\,;\, \mathsf{adv}\ A \triangleright \Delta, S$ finalized if $\Gamma \vdash D \triangleright \Delta$ and $\Gamma, \Delta \vdash A : S$ final

$\Gamma \vdash \quad D\,;\, \mathsf{adv}\ A \triangleright \Delta$ if $\Gamma \vdash D \triangleright \Delta$ and $\Gamma, \Delta \vdash A : S$ and $\Gamma \ni S$ finalized

$\Gamma \vdash \mathsf{call}\langle U\rangle$ ok if $\Gamma \vdash U : S$ and $\Gamma \ni S$ finalized
$\Gamma \vdash \quad z\langle U\rangle$ ok if $\Gamma \vdash U : S$ and $\Gamma \ni z : S$
$\Gamma \vdash \quad \vec{A}\langle U\rangle$ ok if $\Gamma \vdash U : S$ and $\forall i.\, \Gamma \vdash A_i : S$ and $\exists i.\, \Gamma \vdash A_i : S$ final
$\Gamma \vdash \quad D; M$ ok if $\Gamma \vdash D \triangleright \Delta$ and $\Gamma, \Delta \vdash M$ ok

To see what it means for D to be consistent with Γ, observe that in order to avoid error condition (1) above, the first declared advice for any given sort S must be final— that is, it cannot proceed. If it were to proceed, it would be guaranteed that there would be no remaining advice, and a runtime error would result. Once a sort has been finalized, nonfinal advice of that sort may then be declared.

The proof of progress relies on the following properties:

- If $\vdash D \triangleright \Gamma$ and $\Gamma \vdash a : s$, then $D \triangleright a : s$.
- If $\vdash D \triangleright \Gamma$ and $\Gamma \vdash U : S$ and $\vdash P : S \triangleright \_$, then $D \triangleright match(P)(U)$.
- If $\vdash D \triangleright \Gamma$ and $\Gamma \ni S$ finalized and $\Gamma \vdash U : S$, then $D \ni \mathsf{adv}\{\_.P \to \_\}$ such that $D \triangleright match(P)(U)$.

Theorem 5 (Progress). *For any term M, if $\vdash M$ ok, then $M \to M'$ for some M'.*

The proof of preservation relies on a substitution lemma, which in turn relies on *compatibility* between substitutions and typing environments.

Definition 6 (Compatibility). $\Gamma \vdash \sigma \sim \Delta$ if and only if $dom(\sigma) = dom(\Delta)$ and $\forall\, a \in dom(\sigma).\ \Gamma \vdash \sigma(a) : \Delta(a)$. □

For example, if $\sigma = [x := a, y := b]$ then $a : s, b : t \vdash \sigma \sim x : s, y : t$.

Lemma 7 (Compatibility). *If $\vdash D \triangleright \Gamma$ and $\Gamma \vdash U : S$ and $\vdash P : S \triangleright \Delta$ and $D \triangleright match(P)(U) = \sigma$, then $D \vdash \sigma \sim \Delta$.*

Lemma 8 (Substitution). *If $\Gamma, \Delta, z : S \vdash D; M$ ok and $\Gamma \vdash \sigma \sim \Delta$ and $\forall\, A \in \vec{A}.\ \Gamma \vdash A : S$, then $\Gamma \vdash D; M[\sigma, z := \vec{A}]$ ok.*

Theorem 9 (Preservation). *If $\Gamma \vdash M$ ok and $M \to M'$ then $\Gamma \vdash M'$ ok.*

4. A functional language with advice

In this section, we present an extension of the λ-calculus in which functions can be named and advised, in the style of [9, 8], and describe its translation into μABC.

4.1 The source language

Due to space constraints, our presentation is necessarily brief. We give some examples of evaluation of the source language in Section 4.3; for further examples and narrative, see [8], which we follow closely. The language is very expressive. Although declarations are sequential, one can write mutually recursive functions using advice. As shown in [8], this language is also powerful enough to capture imperative features. For example, one can create a reference cell as function which accepts unit and returns the value of the cell; getting the stored value is achieved by calling the function; setting the value is achived by placing advice so that the function returns a different value in future calls.

We annotate each abstraction with its type to facilitate the translation presented in the following subsection. We often elide these annotations.

SYNTAX

$A, B ::= \lambda x.M_T$		Abstractions
$D, E ::= \mathsf{fun}\, f = A \mid \mathsf{adv}\{z.\hat{f} \to A\}$		Declarations
$V, U ::= n \mid \mathsf{unit} \mid A$		Values
$M, N ::= V \mid V\, U \mid z\, U \mid D; M \mid \mathsf{let}\, x : T = M; N$		Terms
$T, S ::= \mathsf{Unit} \mid T \to S$		Types
$\Gamma, \Delta ::= \cdot \mid \Gamma, x : T$		Environments

The language is simply typed; nevertheless non-termination is possible due to the imperative quality of aspects. The typing system need not be concerned with finality of advice, since only function names can be advised and functions themselves cannot proceed.

TYPING $(\Gamma \vDash A : T)$ $(\Gamma \vDash D \triangleright \Delta)$ $(\Gamma \vDash M : T)$

$\Gamma \vDash \lambda x.M_{T\to S} : T \to S$ if $\Gamma, x : T \vDash M : S$

$\Gamma \vDash \mathsf{fun}\, f = A \triangleright f : T$ if $\Gamma, f : T \vDash A : T$
$\Gamma \vDash \mathsf{adv}\{z.\hat{f} \to A\} \triangleright \cdot$ if $\Gamma \vDash f : T$ and $\Gamma, z : T \vDash A : T$

$\Gamma \vDash \mathsf{unit} : \mathsf{Unit}$
$\Gamma \vDash n : T$ if $\Gamma \ni n : T$
$\Gamma \vDash V\, U : S$ if $\Gamma \vDash V : T \to S$ and $\Gamma \vDash U : T$
$\Gamma \vDash z\, U : S$ if $\Gamma \vDash z : T \to S$ and $\Gamma \vDash U : T$
$\Gamma \vDash D; M : T$ if $\Gamma \vDash D \triangleright \Delta$ and $\Gamma, \Delta \vDash M : T$
$\Gamma \vDash \mathsf{let}\, x : T = M; N : S$ if $\Gamma \vDash M : T$ and $\Gamma, x : T \vDash U : S$

Evaluation is defined using *lookup*, notated $\vec{D}(f) = A$. Lookup resolves proceed variables, producing a single abstraction which includes the advice on the function in addition to the function body itself. Lookup is defined using the partial function *body* and total function *advise* with forms $body(\vec{D})(f) = A$ and $advise(\vec{D})(f)(A) = B$.

EVALUATION $(M \Rightarrow N)$

$body(\cdot)(f) =$ undefined
$body(D; \vec{E})(f) = A$ if $D = \mathsf{fun}\, f = A$
$body(D; \vec{E})(f) = body(\vec{E})(f)$ otherwise

$advise(\cdot)(f)(A) = A$
$advise(D; \vec{E})(f)(A) = advise(\vec{E})(f)(B[z := A])$ if $D = \mathsf{adv}\{z.\hat{f} \to B\}$
$advise(D; \vec{E})(f)(A) = advise(\vec{E})(f)(A)$ otherwise

$\vec{D}(f) = advise(\vec{D})(f)(body(\vec{D})(f))$

$\vec{D}; f\, V \Rightarrow \vec{D}; A\, V$ if $\vec{D}(f) = A$
$\vec{D}; (\lambda x.N)\, V \Rightarrow \vec{D}; N[x := V]$
$\vec{D}; \mathsf{let}\, x = V; N \Rightarrow \vec{D}; N[x := V]$
$\vec{D}; \mathsf{let}\, x = M; N \Rightarrow \vec{D}; \mathsf{let}\, x = M'; N$ if $\vec{D}; M \Rightarrow \vec{D}; M'$

4.2 The translation

Our translation of lambda into μABC is based on the translation of lambda into pi, and thus is parameterized with respect to a continuation k. Proceed names must be handled specially, and thus the translation is also parameterized by a list of proceed names, paired with their pointcuts.

$$\vartheta ::= \cdot \mid \vartheta, z : f$$

In the translation, the value unit and both the names and types of the source language are treated as μABC names. We also assume a name T^{-1} for every lambda type T, which is used for continuations. By convention, we use the names k, j, i to stand for continuations.

TRANSLATION $(\langle\!|V|\!\rangle^\vartheta = (D)(n))$ $([\![D]\!]^\vartheta = D)$ $([\![M]\!]^\vartheta_k = M)$

$\langle\!|n|\!\rangle^\vartheta \triangleq (\cdot)(n)$
$\langle\!|\mathsf{unit}|\!\rangle^\vartheta \triangleq (\cdot)(\mathsf{unit})$
$\langle\!|\lambda x.M_{T\to S}|\!\rangle^\vartheta \triangleq (\mathsf{new}\, f : T \to S; \mathsf{adv}\{\hat{f}, x : T, k : S^{-1} \to [\![M]\!]^\vartheta_k\})(f)$
where $f \notin fn(M)$

$[\![\mathsf{fun}\, f = \lambda x.M_{T\to S}]\!]^\vartheta \triangleq \mathsf{new}\, f : T \to S; \mathsf{adv}\{\hat{f}, x : T, k : S^{-1} \to [\![M]\!]^\vartheta_k\}$
$[\![\mathsf{adv}\{z.\hat{f} \to \lambda x.M_{T\to S}\}]\!]^\vartheta \triangleq \mathsf{adv}\{z.\hat{f}, x : T, k : S^{-1} \to [\![M]\!]^{\vartheta, z:f}_k\}$

$[\![V]\!]^\vartheta_k \triangleq D; \mathsf{call}\langle\hat{k}, n\rangle$ where $\langle\!|V|\!\rangle^\vartheta = (D)(n)$

$[\![\mathrm{V\ U}]\!]^{\vartheta}_k \triangleq D\,;\, E\,;\, \mathsf{call}\langle \hat{g}, n, k\rangle$ where $g \notin dom(\vartheta)$
and $(\!|\mathrm{V}|\!)^{\vartheta} = (D)(g)$ and $(\!|\mathrm{U}|\!)^{\vartheta} = (E)(n)$
$[\![z\ \mathrm{U}]\!]^{\vartheta}_k \triangleq E\,;\, z\langle \hat{f}, n, k\rangle$ where $\vartheta(z) = f$ and $(\!|\mathrm{U}|\!)^{\vartheta} = (E)(n)$
$[\![\mathsf{let}\ x\!:\!\mathrm{T} = \mathrm{M}\,;\, \mathrm{N}]\!]^{\vartheta}_k \triangleq \mathsf{new}\ j\!:\!\mathrm{T}^{-1}\,;\, \mathsf{adv}\,\{\hat{j}, x\!:\!\mathrm{T} \to [\![\mathrm{N}]\!]^{\vartheta}_k\}\,;\, [\![\mathrm{M}]\!]^{\vartheta}_j$
where $j \notin fn(\mathrm{M})$
$[\![\mathrm{D}\,;\, \mathrm{M}]\!]^{\vartheta}_k \triangleq [\![\mathrm{D}]\!]^{\vartheta}\,;\, [\![\mathrm{M}]\!]^{\vartheta}_k$

4.3 Examples

Example 10. Consider the λ-calculus term $(\lambda x.x^2)\ 5$. It evaluates as: $(\lambda x.x^2)\ 5 \Rightarrow 25$. The translation of the λ-calculus term into μABC with continuation k is shown below; we show how it evaluates to $\mathsf{call}\langle \hat{k}, 25\rangle$.

$$\begin{aligned}[\![(\lambda x.x^2)\ 5]\!]_k = &\mathsf{new}\, f\,;\\ &\mathsf{adv}\,\{\hat{f}, x, j \to \mathsf{call}\langle \hat{j}, x^2\rangle\}\,;\\ &\mathsf{call}\langle \hat{f}, 5, k\rangle\end{aligned}$$

The name f represents the function $\lambda x.x^2$. The function is implemented using by the advice declaration $\mathsf{adv}\,\{\hat{f}, x, j \to \mathsf{call}\langle \hat{j}, x^2\rangle\}$. The function is applied to the argument 5 by calling function f on argument 5 with continuation k.

First, $\mathsf{call}\langle \hat{f}, 5, k\rangle$ looks up the advice, which gets enqueued on the current event:

$$\to \{\hat{f}, x, j \to \mathsf{call}\langle \hat{j}, x^2\rangle\}\langle f, 5, k\rangle$$

The body of the advice executes with 5 bound to x, and k bound to j:

$$\to \mathsf{call}\langle \hat{k}, 5^2\rangle \qquad \square$$

Example 11. Consider the λ-calculus term $\mathsf{let}\ x = 5\,;\, (\lambda y.y^2)x$. It evaluates as: $\mathsf{let}\ x = 5\,;\, (\lambda y.y^2)x \Rightarrow (\lambda y.y^2)5 \Rightarrow 25$. The translation of the λ-calculus term into μABC with continuation k is shown below; we show how it evaluates to $\mathsf{call}\langle \hat{k}, 25\rangle$.

$$\begin{aligned}[\![\mathsf{let}\ x = 5\,;\, (\lambda y.y^2)x]\!]_k = &\mathsf{new}\ j\,;\\ &\mathsf{adv}\,\{\hat{j}, x \to \mathsf{new}\, f\,;\\ &\qquad \mathsf{adv}\,\{\hat{f}, y, i \to \mathsf{call}\langle \hat{i}, y^2\rangle\}\,;\\ &\qquad \mathsf{call}\langle \hat{f}, x, k\rangle\}\,;\\ &\mathsf{call}\langle \hat{j}, 5\rangle\end{aligned}$$

First, $\mathsf{call}\langle \hat{j}, 5\rangle$ looks up the advice, which gets enqueued on the current event:

$$\to \{\hat{j}, x \to \mathsf{new}\, f\,;\, \mathsf{adv}\,\{\hat{f}, y, i \to \mathsf{call}\langle \hat{i}, y^2\rangle\}\,;\, \mathsf{call}\langle \hat{f}, x, k\rangle\}\langle \hat{j}, 5\rangle$$

The advice body then gets executed with 5 bound to x:

$$\to \mathsf{new}\, f\,;\, \mathsf{adv}\,\{\hat{f}, y, i \to \mathsf{call}\langle \hat{i}, y^2\rangle\}\,;\, \mathsf{call}\langle \hat{f}, 5, k\rangle$$

$\mathsf{call}\langle \hat{f}, 5, k\rangle$ looks up the advice, which gets enqueued on the current event:

$$\to \{\hat{f}, y, i \to \mathsf{call}\langle \hat{i}, y^2\rangle\}\langle \hat{f}, 5, k\rangle$$

The advice body then gets executed with 5 bound to y and k bound to i:

$$\to \mathsf{call}\langle \hat{k}, 5^2\rangle \qquad \square$$

Example 12. Consider the λ-calculus term $\mathsf{fun}\, f = \lambda y.y^2\,;\, \mathsf{let}\ x = 5\,;\, f\ x$. It evaluates as: $\mathsf{fun}\, f = \lambda y.y^2\,;\, \mathsf{let}\ x = 5\,;\, f\ x \Rightarrow \mathsf{let}\ x = 5\,;\, (\lambda y.y^2)x \Rightarrow (\lambda y.y^2)5 \Rightarrow 25$. The translation of the λ-calculus term into μABC with continuation k is shown below; we show how it evaluates to $\mathsf{call}\langle \hat{k}, 25\rangle$.

$$\begin{aligned}[\![\mathsf{fun}\, f = \lambda y.y^2\,;\, \mathsf{let}\ x = 5\,;\, f\ x]\!]_k = &\mathsf{new}\, f\\ &\mathsf{adv}\,\{\hat{f}, y, j \to \mathsf{call}\langle \hat{j}, y^2\rangle\}\\ &\mathsf{new}\ i\,;\\ &\mathsf{adv}\,\{\hat{i}, x \to \mathsf{call}\langle \hat{f}, x, k\rangle\}\,;\\ &\mathsf{call}\langle \hat{i}, 5\rangle\end{aligned}$$

First, $\mathsf{call}\langle \hat{i}, 5\rangle$ looks up the advice, which gets enqueued on the current event:

$$\to \{\hat{i}, x \to \mathsf{call}\langle \hat{f}, x, k\rangle\}\langle \hat{i}, 5\rangle$$

The advice body is executed with 5 bound to x:

$$\to \mathsf{call}\langle \hat{f}, 5, k\rangle$$

$\mathsf{call}\langle \hat{f}, 5, k\rangle$ looks up advice, which gets enqueued on the current event:

$$\to \{\hat{f}, y, j \to \mathsf{call}\langle \hat{j}, y^2\rangle\}\langle \hat{f}, 5, k\rangle$$

The advice body is executed with 5 bound to y and k bound to j:

$$\to \mathsf{call}\langle \hat{k}, 5^2\rangle \qquad \square$$

Example 13. Consider the λ-calculus term

$$\begin{aligned}\mathrm{M} = &\mathsf{fun}\, f = \lambda y.y^2\,;\\ &\mathsf{adv}\,\{z.\hat{f} \to \lambda x.z(x+1)\}\,;\\ &f\ 5\end{aligned}$$

It evaluates as: $\mathrm{M} \Rightarrow (\lambda x.\,(\lambda y.y^2)(x+1))5 \Rightarrow (\lambda y.y^2)(5+1) \Rightarrow (5+1)^2$. The translation of the λ-calculus term into μABC with continuation k is shown below; we show how it evaluates to $\mathsf{call}\langle \hat{k}, (5+1)^2\rangle$.

$$\begin{aligned}[\![\mathrm{M}]\!]_k = &\mathsf{new}\, f\,;\\ &\mathsf{adv}\,\{\hat{f}, y, k \to \mathsf{call}\langle \hat{k}, y^2\rangle\}\,;\\ &\mathsf{adv}\,\{z.\hat{f}, x, k \to z\langle \hat{f}, x+1, k\rangle\}\,;\\ &\mathsf{call}\langle \hat{f}, 5, k\rangle\end{aligned}$$

First, $\mathsf{call}\langle \hat{f}, 5, k\rangle$ looks up the two pieces of advice and enqueues them on the current event:

$$\to \{\hat{f}, y, k \to \mathsf{call}\langle \hat{k}, y^2\rangle\}, \{z.\hat{f}, x, k \to z\langle \hat{f}, x+1, k\rangle\}\langle \hat{f}, 5, k\rangle$$

The body of the newest advice executes, with 5 bound to x, k bound to k, and the remaining advice bound to z:

$$\to \{\hat{f}, y, k \to \mathsf{call}\langle \hat{k}, y^2\rangle\}\langle \hat{f}, 5+1, k\rangle$$

The body of the advice now executes, with $5+1$ bound to y and k bound to k:

$$\to \mathsf{call}\langle \hat{k}, 36\rangle \qquad \square$$

Example 14. Consider the λ-calculus term

$$\begin{aligned}\mathrm{M} = &\mathsf{fun}\, f = \lambda y.y^2\,;\\ &\mathsf{adv}\,\{z.\hat{f} \to \lambda y_1.z(y_1+1)\}\,;\\ &\mathsf{adv}\,\{z.\hat{f} \to \lambda y_2.\mathsf{let}\ x = z(y_2)\,;\, z(x)\}\,;\\ &f\ 5\end{aligned}$$

Let $\mathrm{A} = (\lambda y_1.(\lambda y.y^2)(y_1+1))$. It evaluates as:

$$\begin{aligned}\mathrm{M} &\Rightarrow (\lambda y_2.\mathsf{let}\ x = \mathrm{A}\ y_2\,;\, \mathrm{A}\ x)\ 5\\ &\Rightarrow \mathsf{let}\ x = \mathrm{A}\ 5\,;\, \mathrm{A}\ x\\ &\Rightarrow^* \mathsf{let}\ x = (5+1)^2\,;\, \mathrm{A}\ x\\ &\Rightarrow \mathrm{A}(5+1)^2\\ &\Rightarrow^* ((5+1)^2+1)^2\end{aligned}$$

The translation of the λ-calculus term into μABC with continuation k is shown below; we show how it evaluates to $\mathsf{call}\langle \hat{k}, ((5+$

$1)^2+1)^2\rangle$.

$$\begin{aligned}[\![\mathrm{M}]\!]_k &= D; \mathsf{call}\langle \hat{f}, 5, k\rangle\\ D &= \mathsf{new}\, f\,;\, \mathsf{adv}\, A\,;\, \mathsf{adv}\, B\,;\, \mathsf{adv}\, C\,;\\ A &= \{\hat{f}, y, k \to \mathsf{call}\langle \hat{k}, y^2\rangle\}\\ B &= \{z.\hat{f}, y_1, k \to z\langle \hat{f}, y_1+1, k\rangle\}\\ C &= \{z.\hat{f}, y_2, k \to \mathsf{new}\, j\,;\, \mathsf{adv}\, \{\hat{j}, x \to z\langle \hat{f}, x, k\rangle\}\,;\, z\langle \hat{f}, y_2, j\rangle\}\end{aligned}$$

The $\mathsf{call}\langle \hat{f}, 5, k\rangle$ triggers advice lookup, and enqueues the three matching pieces of advice onto the current event:

$$[\![\mathrm{M}]\!]_k \to D\,;\, (A,B,C)\langle \hat{f}, 5, k\rangle$$

The newest advice body is executed with 5 bound to y_2, k bound to j, and the remaining advice bound to z:

$$\to E\,;\, (A,B)\langle \hat{f}, 5, j\rangle$$

where

$$E = D\,;\, \mathsf{new}\, j\,;\, \mathsf{adv}\, \{\hat{j}, x \to (A,B)\langle \hat{f}, x, k\rangle\}$$

The newest advice body is executed with 5 bound to y_1, j bound to k, and the remaining advice bound to z:

$$\to E\,;\, A\langle \hat{f}, 5+1, j\rangle$$

The remaining advice body is executed with $5+1$ bound to y and j bound to k:

$$\to E\,;\, \mathsf{call}\langle \hat{j}, (5+1)^2\rangle$$

The $\mathsf{call}\langle \hat{j}, (5+1)^2\rangle$ triggers another advice lookup, and enqueues the matching advice onto the current event:

$$\to E\,;\, \{\hat{j}, x \to (A,B)\langle f, x, k\rangle\}\langle \hat{j}, (5+1)^2\rangle$$

The advice body is executed with $(5+1)^2$ bound to x:

$$\to E\,;\, (A,B)\langle \hat{f}, (5+1)^2, k\rangle$$

The newest advice body is executed with $(5+1)^2$ bound to y_1 and k bound to k:

$$\to E\,;\, (A)\langle \hat{f}, (5+1)^2+1, k\rangle$$

The remaining advice body is executed with $(5+1)^2+1$ bound to y and k bound to k:

$$\to E\,;\, \mathsf{call}\langle \hat{k}, ((5+1)^2+1)^2\rangle \qquad \square$$

4.4 Correctness

Our correctness proof is stated modulo a "structural congruence" on μABC terms. Most of the axioms defining this congruence are innocuous, but the last, *unrolling*, is stated in terms of the translation defined above. As stated in the introduction, there is a further problem in this approach: the congruence is not justified by any semantic reasoning. Nonetheless, our intention is to define this relation such that two structurally equivalent terms in effect "behave the same way".

We provide short examples demonstrating each of the structural equivalence rules, followed by a formal statement of the rules.

Example 15 (Hoisting). Hoisting enables us to move declarations out of the body of an advice declaration, provided that none of the variables in the declarations are bound in the advice body. For instance, the μABC term

$$\begin{array}{l}\mathsf{new}\, f\,;\\ \mathsf{adv}\, \{\hat{f}, y, k \to \mathsf{new}\, g\,;\, \mathsf{adv}\, \{\hat{g}, x, j \to \mathsf{call}\langle \hat{g}, y+1, k\rangle\}\,;\, \mathsf{call}\langle \hat{g}, y, k\rangle\}\\ \mathsf{call}\langle \hat{f}, 10, k\rangle\end{array}$$

is structurally equivalent to

$$\begin{array}{l}\mathsf{new}\, g\,;\, \mathsf{adv}\, \{\hat{g}, x, j \to \mathsf{call}\langle \hat{g}, y+1, k\rangle\}\,;\\ \mathsf{new}\, f\,;\, \mathsf{adv}\, \{\hat{f}, y, k \to \mathsf{call}\langle \hat{g}, y, k\rangle\}\\ \mathsf{call}\langle \hat{f}, 10, k\rangle\end{array}$$

"Hoisting" g's name and advice declaration out of f's advice declaration should have no effect on how the term evaluates. $\square$

Example 16 (Reordering). Reordering says that two declarations D and E can be swapped, so long as $fn(E) \not\in bn(D)$, and vice versa. For instance:

$$\begin{array}{ll}\mathsf{new}\, f\,; & \mathsf{new}\, g\,;\\ \mathsf{adv}\, \{\hat{f}, x, k \to \mathsf{call}\langle \hat{k}, x\rangle\}\,; & \mathsf{adv}\, \{\hat{g}, x, k \to \mathsf{call}\langle \hat{k}, x\rangle\}\,;\\ \mathsf{new}\, g\,; & \equiv \mathsf{new}\, f\,;\\ \mathsf{adv}\, \{\hat{g}, x, k \to \mathsf{call}\langle \hat{k}, x\rangle\}\,; & \mathsf{adv}\, \{\hat{f}, x, k \to \mathsf{call}\langle \hat{k}, x\rangle\}\,;\\ \mathsf{call}\langle \hat{f}, 10, k\rangle & \mathsf{call}\langle \hat{f}, 10, k\rangle\end{array}$$

Whether f's name and advice is declared before or after g's is irrelevant to how the term evaluates. The following, however, is not allowed, however, since $\mathsf{new}\, f$ must be declared before f can appear in an advice declaration:

$$\begin{array}{ll}\mathsf{new}\, f\,; & \mathsf{adv}\, \{\hat{f}, x, k \to \mathsf{call}\langle \hat{k}, x\rangle\}\,;\\ \mathsf{adv}\, \{\hat{f}, x, k \to \mathsf{call}\langle \hat{k}, x\rangle\}\,; & \not\equiv \mathsf{new}\, f\,;\\ \mathsf{call}\langle \hat{f}, 10, k\rangle & \mathsf{call}\langle \hat{f}, 10, k\rangle\end{array} \qquad \square$$

Example 17 (Garbage Collection). Garbage collection allows us to eliminate "dead" declarations. For instance, in the following term:

$$\begin{array}{l}\mathsf{new}\, f\,;\, \mathsf{adv}\, \{\hat{f}, x, k \to \mathsf{call}\langle \hat{k}, x^2\rangle\}\,;\\ \mathsf{new}\, g\,;\, \mathsf{adv}\, \{\hat{g}, x, k \to \mathsf{call}\langle \hat{k}, x+1\rangle\}\,;\\ \mathsf{call}\langle \hat{f}, 10, k\rangle\end{array}$$

The two declarations $\mathsf{new}\, g$ and $\mathsf{adv}\, \{\hat{g}, x, k \to \mathsf{call}\langle \hat{k}, x+1\rangle\}$ are never used, and as such, can be eliminated without affecting how the term evaluates. Thus the above term is structurally equivalent to

$$\mathsf{new}\, f\,;\, \mathsf{adv}\, \{\hat{f}, x, k \to \mathsf{call}\langle \hat{k}, x^2\rangle\}\,;\, \mathsf{call}\langle \hat{f}, 10, k\rangle \qquad \square$$

Example 18 (Unrolling). Translating function applications from λ-calculus into μABC is extremely intricate. For instance, consider the λ-calculus term

$$\begin{array}{l}\mathsf{fun}\, f = \lambda x.x\,;\\ \mathsf{adv}\, \{z_1.\hat{f} \to \lambda y.z_1\langle y^2\rangle\}\,;\\ \mathsf{adv}\, \{z_2.\hat{f} \to \lambda w.z_2\langle w+1\rangle\}\,;\\ f\ 5\end{array}$$

The translation of this term, with continuation k, is

$$D\,;\, \mathsf{call}\langle \hat{f}, 5, k\rangle$$

where

$$\begin{aligned}D &= \mathsf{new}\, f\,;\, \mathsf{adv}\, A\,;\, \mathsf{adv}\, B\,;\, \mathsf{adv}\, C\\ A &= \{\hat{f}, x, k \to \mathsf{call}\langle \hat{k}, x\rangle\}\\ B &= \{z_1.\hat{f}, y, k \to z_1\langle f, y^2, k\rangle\}\\ C &= \{z_2.\hat{f}, w, k \to z_2\langle f, w+1, k\rangle\}\end{aligned}$$

The λ-calculus term evaluates to

$$(\lambda w.(\lambda y.(\lambda x.x)\ y^2)\ (w+1))\ 5$$

the translation of which is

$$\begin{array}{l}\mathsf{new}\, h\,;\\ \mathsf{adv}\, \{\hat{h}, w, i \to \mathsf{new}\, f\,;\\ \qquad \mathsf{adv}\, \{\hat{f}, y, k \to \mathsf{new}\, g\,;\\ \qquad\qquad \mathsf{adv}\, \{\hat{g}, x, j \to \mathsf{call}\langle \hat{j}, x\rangle\}\,;\\ \qquad\qquad \mathsf{call}\langle \hat{g}, y^2, k\rangle\}\,;\\ \qquad \mathsf{call}\langle \hat{f}, w+1, i\rangle\}\,;\\ \mathsf{call}\langle \hat{h}, 5, k\rangle\end{array}$$

The fundamental difficulty arises from the fact that the translation of the original λ-calculus term yields a string of corresponding

advice declarations, while the translation of the λ-calculus term after one step yields a single nested advice declaration. In this vein, unrolling allows us to expand the rolled form into the following structurally equivalent term:

$$D\,;\,(A,B,C)\langle \hat{f},5,k\rangle \qquad \square$$

STRUCTURAL EQUIVALENCE $(M \equiv N)$

Hoisting:
$D\,;\,\mathsf{adv}\{z.P \to E\,;\,M\} \equiv D\,;\,E\,;\,\mathsf{adv}\{z.P \to M\}$ if $z, bn(P) \notin fn(E)$

Reordering:
$D\,;\,E\,;\,M \equiv E\,;\,D\,;\,M$ if $fn(E) \notin bn(D)$ and $fn(D) \notin bn(E)$

Garbage Collection:
$D\,;\,\mathsf{new}\,f\,;\,\mathsf{adv}\{z.P,\hat{f},Q \to M\}\,;\,N \equiv D\,;\,N$ if $f \notin fn(N)$

Unrolling:
$$D\,;\,\mathsf{call}\langle \hat{f},v,k\rangle \equiv D\,;\,(\{z_0.\hat{f},x_0,k_0 \to [\![\mathrm{L}_0]\!]_{k_0}\},$$
$$\vdots$$
$$\{z_m.\hat{f},x_m,k_m \to [\![\mathrm{L}_m]\!]_{k_m}\})\langle \hat{f},v,k\rangle\rangle$$
where $D = \mathsf{new}\,f\,;$
$$\mathsf{adv}\{z_m.\hat{f},x_m,k \to [\![\mathrm{L}_m\sigma_m]\!]_k\}\,;$$
and $\sigma_0 = [\,]$ and $\sigma_n = [z_n := \lambda y_{n-1}.\mathrm{L}_{n-1}\sigma_{n-1}]$

The following theorem states that our translation from λ-calculus into μABC is correct up to structural congruence.

Lemma 19 (Substitution).

$$[\![\mathrm{M}[x := \mathrm{V}]]\!]^{\vartheta}_k \equiv D\,;\,[\![\mathrm{M}]\!]^{\vartheta}_k[x := v]$$
$$(\!|\mathrm{U}[x := \mathrm{V}]|\!)^{\vartheta} \equiv (D\,;\,E[x := v])(u)$$

where $(\!|\mathrm{V}|\!)^{\vartheta} = (D)(v)$ *and* $(\!|\mathrm{U}|\!)^{\vartheta} = (E)(u)$ $\square$

Proposition 20. *If* $\mathrm{M} \Rightarrow \mathrm{N}$, *then* $[\![\mathrm{M}]\!]_c \to^* \equiv [\![\mathrm{N}]\!]_c$.

5. A Small Object Language

We give a translation into μABC of a small, object-oriented language with advice.

The source language is based roughly on that of [3, 7], but there are a few differences. First, the evaluation strategy is based on the evaluation strategy for lambda calculus presented in the last section; in particular the definition of lookup. Secondly, we ignore fields for simplicity.

As before, z ranges over proceed names. In addition, we use the following conventions for names.

- k, j, i range over continuations,
- ℓ, m, n range over method names,
- a, b, e range over class names,
- p, q, x, y, v, u range over object names,
  - p, q range over proper object names,
  - x, y range over variable names,
  - v, u range over variables or proper object names,

OBJECT CALCULUS

$\mathbf{A},\mathbf{B} ::= \lambda\vec{x}.\mathbf{M}_{\mathbf{T}}$	Abstractions
$\mathbf{C} ::= \mathsf{cls}\,a : b\,\{\bar{\ell} = \bar{\mathbf{A}}\}$	Class Declarations
$\mathbf{D},\mathbf{E} ::= \mathsf{obj}\,p : a \mid \mathsf{advc}\{z.a.\hat{\ell} \to \mathbf{A}\}$	
$\mathbf{M},\mathbf{N} ::= v \mid v.\ell(\vec{u}) \mid z(\vec{u}) \mid \mathbf{A}(\vec{u}) \mid \mathbf{D}\,;\,\mathbf{M} \mid \mathsf{let}\,x : a = \mathbf{M}\,;\,\mathbf{N}$	
$\mathbf{T},\mathbf{S} ::= \vec{a} \to b$	Method Types
$\Gamma,\Delta ::= \cdot \mid \Gamma, x : a$	Environments

As usual [6], we fix a class table $\bar{\mathbf{C}}$; we assume that Object is not declared and that the induced subclass relation is antisymmetric, with greatest element Object. (The subclass relation is the smallest preorder on class names induced by the rule: $a \le b$ if $\bar{\mathbf{C}} \ni \mathsf{cls}\,a : b\,\{\cdots\}$.) We also assume that every declaration in the class table is well typed (ie, $\forall \mathbf{C} \in \bar{\mathbf{C}}.\ \Vdash \mathbf{C}\ \mathsf{ok}$).

The function *body* is used in both evaluation and typing. We leave out irrelevant bits.

$(body(a.\ell) = \mathbf{A} : \mathbf{T})$

$$body(a.\ell_i) = \mathbf{A}_i : \mathbf{T}_i \text{ if } \bar{\mathbf{C}}(a) = \mathsf{cls}\,a : b\,\{\bar{\ell} = \bar{\mathbf{A}}\} \text{ and } \mathbf{A}_i = \lambda\vec{x}.\mathbf{M}_{\mathbf{T}_i}$$
$$body(a.\ell) = \mathbf{A} : \mathbf{T} \text{ if } \bar{\mathbf{C}}(a) = \mathsf{cls}\,a : b\,\{\bar{m} = \bar{\mathbf{B}}\} \text{ and } \ell \notin \bar{m} \text{ and } body(b.\ell) = \mathbf{A} : \mathbf{T}$$

TYPING $(\Gamma \Vdash \mathbf{A} : \mathbf{T})$ $(\Vdash \mathbf{C}\ \mathsf{ok})$ $(\Gamma \Vdash \mathbf{D} \triangleright \Gamma')$ $(\Gamma \Vdash \mathbf{M} : a)$

$$\Gamma \Vdash \lambda\vec{x}.\mathbf{M}_{\vec{a}\to b} : \vec{a}\to b \text{ if } \Gamma,\vec{x} : \vec{a} \Vdash \mathbf{M} : b' \text{ and } b' \le b$$
$$\Vdash \mathsf{cls}\,a : b\,\{\bar{\ell} = \bar{\mathbf{A}}\}\ \mathsf{ok} \text{ if } \forall i.\ \mathsf{self} : a \Vdash \mathbf{A}_i : \mathbf{T}_i \text{ and } \forall i.\ body(b.\ell_i) = \mathbf{S} \text{ implies } \mathbf{T}_i = \mathbf{S}$$
$$\Gamma \Vdash \mathsf{obj}\,p : a \triangleright p : a \text{ if } \bar{\mathbf{C}}(a) \text{ defined}$$
$$\Gamma \Vdash \mathsf{advc}\{z.a.\hat{\ell} \to \mathbf{A}\} \triangleright \cdot \text{ if } body(a.\ell) = \mathbf{T} = \_ \xrightarrow{c} \_ \text{ and } \Gamma,\mathsf{self} : a, z : \mathbf{T} \Vdash \mathbf{A} : \mathbf{T}$$

$\Gamma \Vdash v : a$	if $\Gamma \ni v : a$
$\Gamma \Vdash v.\ell(\vec{u}) : b$	if $\Gamma \Vdash v : e$ and $body(e.\ell) = \vec{a}'\to b$ and $\forall i.\ \Gamma \Vdash u_i : a_i$ and $a_i \le a'_i$
$\Gamma \Vdash z(\vec{u}) : b$	if $\Gamma \Vdash z : \vec{a}'\to b$ and $\forall i.\ \Gamma \Vdash u_i : a_i$ and $a_i \le a'_i$
$\Gamma \Vdash \mathbf{A}(\vec{u}) : b$	if $\Gamma \Vdash \mathbf{A} : \vec{a}'\to b$ and $\forall i.\ \Gamma \Vdash u_i : a_i$ and $a_i \le a'_i$
$\Gamma \Vdash \mathbf{D}\,;\,\mathbf{M} : a$	if $\Gamma \Vdash \mathbf{D} \triangleright \Delta$ and $\Gamma,\Delta \Vdash \mathbf{M} : a$
$\Gamma \Vdash \mathsf{let}\,x : a = \mathbf{M}\,;\,\mathbf{N} : \mathbf{S}$	if $\Gamma \Vdash \mathbf{M} : a'$ and $a' \le a$ and $\Gamma, x : \mathbf{T} \Vdash \mathbf{N} : \mathbf{S}$

The functions for advising and lookup now have have the form $advise(\vec{\mathbf{D}})(p : a.\ell)(\mathbf{A}) = \mathbf{B}$ and $\vec{\mathbf{D}}(p.\ell) = \mathbf{A}$.

EVALUATION $(\vec{\mathbf{D}}\,;\,\mathbf{M} \Rightarrow \vec{\mathbf{E}}\,;\,\mathbf{N})$

$$advise(\cdot)(p : a.\ell)(\mathbf{A}) = \mathbf{A}$$
$$advise(\mathbf{D}\,;\,\vec{\mathbf{E}})(p : a.\ell)(\mathbf{A}) = \begin{cases} advise(\vec{\mathbf{E}})(p : a.\ell)(\mathbf{B}[z := \mathbf{A}]) & \text{if } \mathbf{D} = \mathsf{advc}\{z.a'.\hat{\ell} \to \mathbf{B}\} \text{ and } a \le a' \\ advise(\vec{\mathbf{E}})(p : a.\ell)(\mathbf{A}) & \text{otherwise} \end{cases}$$
$$\vec{\mathbf{D}}(p.\ell) = advise(\vec{\mathbf{D}})(p : a.\ell)(body(a.\ell)) \text{ if } \vec{\mathbf{D}} \ni p : a$$
$$\vec{\mathbf{D}}\,;\,p.\ell(\vec{q}) \Rightarrow \vec{\mathbf{D}}\,;\,(\mathbf{A}[\mathsf{self} := p])(\vec{q}) \text{ if } \vec{\mathbf{D}}(p.\ell) = \mathbf{A}$$
$$\vec{\mathbf{D}}\,;\,(\lambda x.\mathbf{N})(\vec{q}) \Rightarrow \vec{\mathbf{D}}\,;\,\mathbf{N}[x := \vec{q}]$$
$$\vec{\mathbf{D}}\,;\,\mathsf{let}\,x = p\,;\,\mathbf{N} \Rightarrow \vec{\mathbf{D}}\,;\,\mathbf{N}[x := p]$$
$$\vec{\mathbf{D}}\,;\,\mathsf{let}\,x = \mathbf{M}\,;\,\mathbf{N} \Rightarrow \vec{\mathbf{D}}\,;\,\mathsf{let}\,x = \mathbf{M}'\,;\,\mathbf{N} \quad \text{if } \vec{\mathbf{D}}\,;\,\mathbf{M} \Rightarrow \vec{\mathbf{D}}\,;\,\mathbf{M}'$$

As before, the translation of terms is parameterized with respect to continuations and bound proceed names. In this case proceed names are bound to pointcuts of a different shape than in the functional case.

$$\vartheta ::= \cdot \mid \vartheta, z : \langle p,\ell\rangle$$

TRANSLATION $([\![\mathbf{C}]\!] = D)$ $([\![\ell = \mathbf{A}]\!]^a = D)$

$$[\![\mathsf{cls}\,a : b\,\{\bar{\ell} = \bar{\mathbf{A}}\}]\!] \triangleq \mathsf{new}\,a : b\,;\,\mathsf{new}\,\vec{m}\,;\,[\![\bar{\ell} = \bar{\mathbf{A}}]\!]^a \text{ where } \vec{m} = (\ell_i \mid body(b.\ell_i) \text{ undefined})$$
$$[\![\ell = \lambda\vec{x}.\mathbf{M}_{\vec{a}\to b}]\!]^e \triangleq \mathsf{adv}\{\mathsf{self} : e, \hat{\ell}, \vec{x} : \vec{a}, k : b^{-1} \to [\![\mathbf{M}]\!]^{\vartheta}_k\}$$

TRANSLATION $(\!|\mathbf{A}|\!)^{\vartheta} = (D)(f))$ $([\![\mathbf{D}]\!]^{\vartheta} = D)$ $([\![\mathbf{M}]\!]_k^{\vartheta} = M)$

$$(\!|\lambda\vec{x}.\mathbf{M}_{\vec{a}\to b}|\!)^{\vartheta} \triangleq (\mathsf{new}\, f:\vec{a}\to b\,;\, \mathsf{adv}\{\hat{f},\vec{x}:\vec{a},k:b^{-1}\to [\![\mathbf{M}]\!]_k^{\vartheta}\})(f) \quad \text{where } f\notin fn(\mathbf{M})$$

$$[\![\mathsf{obj}\, p:a]\!]^{\vartheta} \triangleq \mathsf{new}\, p:a$$

$$[\![\mathsf{advc}\{z.e.\hat{\ell}\to\lambda\vec{x}.\mathbf{M}_{\vec{a}\to b}\}]\!]^{\vartheta} \triangleq \mathsf{adv}\{z.\mathsf{self}:e,\hat{\ell},\vec{x}:\vec{a},k:b^{-1}\to [\![\mathbf{M}]\!]_k^{\vartheta,z:\langle \mathsf{self},\hat{\ell}\rangle}\}$$

$$[\![v]\!]_k^{\vartheta} \triangleq \mathsf{call}\langle\hat{k},v\rangle$$

$$[\![v.\ell(\vec{u})]\!]_k^{\vartheta} \triangleq \mathsf{call}\langle v,\hat{\ell},\vec{u},k\rangle$$

$$[\![z(\vec{u})]\!]_k^{\vartheta} \triangleq z\langle p,\hat{\ell},\vec{u},k\rangle \text{ where } \vartheta(z)=\langle p,\ell\rangle$$

$$[\![\mathbf{A}(\vec{u})]\!]_k^{\vartheta} \triangleq D\,;\,\mathsf{call}\langle\hat{g},\vec{u},k\rangle \text{ where } (\!|\mathbf{A}|\!)^{\vartheta} = (D)(g)$$

$$[\![\mathbf{D}\,;\mathbf{M}]\!]_k^{\vartheta} \triangleq [\![\mathbf{D}\,;\mathbf{M}]\!]_k^{\vartheta} \triangleq [\![\mathbf{D}]\!]^{\vartheta}\,;\,[\![\mathbf{M}]\!]_k^{\vartheta}$$

$$[\![\mathsf{let}\, x:a=\mathbf{M}\,;\,\mathbf{N}]\!]_k^{\vartheta} \triangleq \mathsf{new}\, j:a^{-1}\,;\,\mathsf{adv}\{\hat{j},x:a\to [\![\mathbf{N}]\!]_k^{\vartheta}\}\,;\,[\![\mathbf{M}]\!]_j^{\vartheta}$$

We begin with a simple example.

Example 21 (Methods). Consider the following program fragment in the class-based language.

$$\mathsf{cls}\, a\{\ell=\lambda x.x^2\}\,;\,\mathsf{obj}\, p:a\,;\, p.\ell(5)$$

The class-based term evaluates as follows:

$$\begin{array}{l}\Rightarrow \mathsf{cls}\, a\{\ell=\lambda x.x^2\}\,;\,\mathsf{obj}\, p:a\,;\,(\lambda x.x^2)\ 5\\ \Rightarrow \mathsf{cls}\, a\{\ell=\lambda x.x^2\}\,;\,\mathsf{obj}\, p:a\,;\,25\end{array}$$

The translation of the original term into μABC yields:

$$\begin{array}{l}\mathsf{new}\, a\,;\,\mathsf{adv}\{\mathsf{self}:a,\hat{\ell},x,k\to\mathsf{call}\langle\hat{k},x^2\rangle\}\,;\\ \mathsf{new}\, p:a\,;\,\mathsf{call}\langle p,\hat{\ell},5,k\rangle\,;\end{array}$$

Observe that the μABC term evaluates to $\mathsf{call}\langle\hat{k},25\rangle$:

$$\begin{array}{l}\to \mathsf{new}\, a\,;\,\mathsf{adv}\{\mathsf{self}:a,\hat{\ell},x,k\to\mathsf{call}\langle\hat{k},x^2\rangle\}\,;\\ \quad \mathsf{new}\, p:a\,;\,\{\mathsf{self}:a,\hat{\ell},x,k\to\mathsf{call}\langle\hat{k},x^2\rangle\}\langle p,\hat{\ell},5,k\rangle\\ \to \mathsf{new}\, a\,;\,\mathsf{adv}\{\mathsf{self}:a,\hat{\ell},x,k\to\mathsf{call}\langle\hat{k},x^2\rangle\}\,;\\ \quad \mathsf{new}\, p:a\,;\,\mathsf{call}\langle\hat{k},5^2\rangle\end{array}$$ □

Example 22 (Methods). Consider the following program fragment in the class-based language.

$$\begin{array}{l}\mathsf{cls}\, a\{\ell=\lambda.M\,;\,m=\lambda.N\}\,;\\ \mathsf{cls}\, b:a\{m=\lambda.P\,;\,n=\lambda.Q\}\,;\\ \mathsf{obj}\, p:b\,;\\ \mathsf{let}\, x=p.\ell()\,;\,\mathsf{let}\, y=p.m()\,;\,\mathsf{let}\, z=p.n()\,;\,U\end{array}$$

Its μABC translation is as follows:

$$\begin{array}{l}\mathsf{new}\, a\,;\,\mathsf{adv}\{\mathsf{self}:a,\hat{\ell},k\to[\![M]\!]\}\,;\,\mathsf{adv}\{\mathsf{self}:a,\hat{m},k\to[\![N]\!]\}\,;\\ \mathsf{new}\, b:a\,;\,\mathsf{adv}\{\mathsf{self}:b,\hat{m},k\to[\![P]\!]\}\,;\,\mathsf{adv}\{\mathsf{self}:b,\hat{n},k\to[\![Q]\!]\}\,;\\ \mathsf{new}\, p:b\,;\\ \mathsf{new}\, k_1\,;\\ \mathsf{adv}\{\hat{k}_1,x\to\mathsf{new}\, k_2\,;\,\mathsf{adv}\{\hat{k}_2,y\to\mathsf{new}\, k_3\,;\,\mathsf{adv}\{\hat{k}_3,z\to[\![U]\!]\}\,;\\ \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{call}\langle p,\hat{n},k_3\rangle\}\,;\\ \qquad\qquad \mathsf{call}\langle p,\hat{m},k_2\rangle\}\,;\\ \mathsf{call}\langle p,\hat{\ell},k_1\rangle\end{array}$$ □

Conjecture 23. *If* $\mathbf{M}\Rightarrow\mathbf{N}$, *then* $[\![\mathbf{M}]\!]_k \to^{*}\equiv [\![\mathbf{N}]\!]_k$. □

6. Conclusions

We have reported some preliminary steps toward attaining a useful type system for μABC. Several challenges remain, all discussed in the introduction. First, we face the niggling difference in substitution times for our source and target calculi. Second, and more interestingly, we require a useful notion of semantic equivalence. Third, and most importantly, we must parameterize the sorting system given here with the type of protocols on names described in the introduction. All these problems have solutions, and the solution to the third promises to be very interesting.

References

[1] M. Abadi, L. Cardelli, P. L. Curien, and J. J. Levy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[2] Glen Bruns, Radha Jagadeesan, Alan Jeffrey, and James Riely. μABC: A minimal aspect calculus. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR 2004: Concurrency Theory*, volume 3170 of *Lecture Notes in Computer Science*, pages 209–224, London, August 2004. Springer.

[3] C. Clifton and G. T. Leavens. Minimao1: An imperative core language for studying aspect-oriented reasoning. *Science of Computer Programming*, 63(3):321–374, 2006.

[4] Christopher Dutchyn, David B. Tucker, and Shriram Krishnamurthi. Semantics and scoping of aspects in higher-order languages. *Science of Computer Programming*, 63(3), 2006.

[5] Peter Hui and James Riely. Temporal aspects as security automata. In *Foundations of Aspect-Oriented Languages (FOAL)*, pages 19–28, 2006.

[6] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

[7] Radha Jagadeesan, Alan Jeffrey, and James Riely. Typed parametric polymorphism for aspects. *Science of Computer Programming*, 63(3), 2006.

[8] Radha Jagadeesan, Corin Pitcher, and James Riely. Open bisimulation for aspects. In Oege de Moor, editor, *AOSD*. ACM, 2007.

[9] Jay Ligatti, David Walker, and Steve Zdancewic. A type-theoretic interpretation of pointcuts and advice. *Science of Computer Programming*, 63(3), 2006.

[10] Davide Sangiorgi and David Walker. *The pi-calculus: a Theory of Mobile Processes*. Cambridge Universtity Press, 1991.

[11] David Walker, Steve Zdancewic, and Jay Ligatti. A theory of aspects. In Colin Runciman and Olin Shivers, editors, *ICFP*, pages 127–139. ACM, 2003.

Towards a Type System for Detecting Never-Matching Pointcut Compositions

Tomoyuki Aotani
Graduate School of Arts and Sciences
University of Tokyo
aotani@graco.c.u-tokyo.ac.jp

Hidehiko Masuhara
Graduate School of Arts and Sciences
University of Tokyo
masuhara@acm.org

ABSTRACT

Pointcuts in the current AspectJ family of languages are loosely checked because the languages allow compositions of pointcuts that never match any join points, which developers are unlikely to intend, for example, `set(* *)&&get(* *)`. We formalize the problem by defining well-formedness of pointcuts and design a novel type system for assuring well-formedness. The type of pointcuts is encoded by using record, union and the bottom types.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications—*Languages*; F.3.3 [**Logics and Meanings of Programs**]: Studies of program constructs—*Type structure*

General Terms

Design, languages, theory

Keywords

AOP, pointcut compositions, records

1. INTRODUCTION

Join point selection mechanisms, i.e., pointcuts, play an important role in AspectJ family of languages such as AspectJ [13] and JBoss AOP. While there have been studies targeting many facets of those languages, such as expressiveness and robustness [1,5,9,12,17], safe pointcut compositions have been less investigated [4, 16]. The property becomes more important the more aspects use composed pointcuts.

This position paper focuses on safe pointcut composability so that composed pointcuts can match at least one join point in *some* program. We call such a pointcut *well-formed*. By checking well-formedness of every pointcuts in aspect definitions, developers can notice unintended pointcut compositions before applying aspects to programs. This property helps programmers to avoid pointcuts that never have any match by merely examining aspect definitions. It is particularly important for separate compilation of aspects. In other words, we are interested in detecting *never-matching* pointcuts compositions whose resulting pointcuts match no join point in *any program*. Note that current AspectJ compiler implementations (abc [2]) can report, only after weaving aspects into classes, that an advice declaration is not woven into any join point shadows. When we separately compile aspects (which is also required by the load-time weaving techniques), the current compilers silently pass never-matching pointcut compositions.

To detect such never-matching pointcut compositions, we are going to develop a type system that guarantees the *well-formedness* of pointcuts, defined as follows:

DEFINITION 1 (WELL-FORMED POINTCUT). *Let U be the set of all well-typed base programs and $JP(b)$ be the set of join points in any execution of a well-typed base program b. Pointcut p is well-formed when it satisfies:*

$$\exists b \in U. \exists j \in JP(b). match(p, j)$$

In other words, a pointcut p is well-formed when there exists a well-typed program that generates a join point matching p.

The rest of the paper is organized as follows. Section 2 explains the problems we address. Section 3 shows a sketch of our type system. Section 4 discusses related work. Section 5 concludes the paper.

2. NEVER-MATCHING POINTCUT COMPOSITIONS

In order to clarify the problems that we address, we present an example in which a pointcut never matches any join point.

AspectJ compilers allow meaningless pointcut compositions that never match join points in any program.

In AspectJ, one can compose any two pointcuts with `&&` (`and`) and `||` (`or`) pointcut designators. For example, one can capture both get and set join points by composing a get and set pointcuts with an `or` operator; i.e., `get(* *)||set(* *)`.

On the other hand, the composition of these two pointcuts with an `and` operator, i.e., `get(* *)&&set(* *)` is meaningless because no join point is get and set at the same time.

3. A SKETCH OF OUR TYPE SYSTEM FOR POINTCUTS

Sixth International Workshop on Foundations of Aspect-Oriented Languages (FOAL 2007), March 13, 2007, Vancouver, BC, Canada.

Table 1: Elaborated pointcut primitives.

	get	set
instance variables	`mget`	`mset`
class variables	`sget`	`sset`

This section gives a sketch of our type system that types pointcuts with respect to the well-formedness. We formalize the system by adding a pointcut and advice mechanism to Featherweight Java [10].

The key idea is to represent the type of join points as a record type. The type of pointcuts is also a record type because a pointcut can be seen as a set of matching join points.

Because the paper focuses on pointcut compositions, we explain only the pointcut types below.

3.1 Overview of Pointcut Types

The pointcut type P is defined as follows.

$$\begin{array}{lcl} P & ::= & \{l_i : {Y_i}^{i\in 1\cdots n}\} \mid P + P \mid \bot \\ Y & ::= & k \mid T^g \mid [{T_i}^{i\in 1\cdots n}] \mid \bullet \\ k & ::= & \texttt{mset} \mid \texttt{mget} \mid \texttt{sset} \mid \texttt{sget} \mid \cdots \\ g & ::= & \bullet \mid \epsilon \end{array}$$
$$T \in \mathit{idpats}$$

Basically, a pointcut type is encoded into a record type $\{l_i : {Y_i}^{i\in 1\cdots n}\}$ that contains most attributes of the `JoinPoint` object in AspectJ such as the kind of the matching join point. The label l corresponds to the attribute of a join point including `this`, `args` and `kind`, and is associated with an attribute type Y. The attribute type Y is either the kind of matching join points k, an element of *idpats* with runtime value availability tag g, or a sequence of the elements of *idpats*. $P + P$ denotes the union of two pointcut types and $\bot$ denotes pointcuts never matching any join point. We assume that $\bot + P$ and $P + \bot$ are equivalent to P. The set *idpats* is the union of the three sets: the singleton set of an $*$, the set of names of primitive types such as `int` and `boolean`, and the set of valid identifiers with respect to the Java Language Specification [8] such as `Object`, `List` and `width`. $[{T_i}^{i\in 1\cdots n}]$ represents a comma-separated sequence of the elements of *idpats*. The single $\bullet$ denotes absence. For example, $\{\texttt{args} : \bullet\}$ denotes that matched join points never have `args` values. Meta-variable k ranges over the kinds of join points such as `get` and `set`. Meta-variable g ranges over runtime value availability tags. When a label l is associated with $T^\bullet$, it denotes that there is no runtime value for the attribute l. For example, $\{\texttt{target} : T^\bullet\}$ represents that the `target` attribute of matching join points is constrained to have the type T but has no runtime value. For readability, we omit the availability tag when it is ϵ, so we simply write T rather than T^ϵ.

3.2 Pointcut Sublanguage and Typing Rules

Since we are still working on details of the type system, the paper demonstrates how pointcuts are typed by merely using `set`, `get`, `args`, `||` and `&&` pointcuts. The pointcut sublanguage is defined as Figure 2. The `args` pointcut does not bind any variable because we are only interested in the pointcut compositions. Instead, an `args` pointcut limits the types and numbers of arguments of matching join points.

We divide `set` and `get` pointcuts into the four pointcuts as is shown in Table 1 so that they explicitly distinguish whether matching join points access the `static` fields or not. This is because the join points related to class fields have no `target` value.[1]

$$\begin{array}{lcl} \mathit{pc} & ::= & \mathit{prm}(T\ C.f) \mid \texttt{args}({T_i}^{i\in 1\cdots n}) \mid \mathit{pc}\&\&\mathit{pc} \mid \mathit{pc}||\mathit{pc} \\ \mathit{prm} & ::= & \texttt{mset} \mid \texttt{sset} \mid \texttt{mget} \mid \texttt{sget} \end{array}$$

Figure 2: Pointcut sublanguage ($T, C, f \in \mathit{idpats}$).

$$\frac{\mathit{pc}_1 : P_1 \quad \mathit{pc}_2 : P_2 \quad P_1 \otimes P_2 \rightsquigarrow P}{\mathit{pc}_1\&\&\mathit{pc}_2 : P}$$

$$\frac{\mathit{pc}_1 : P_1 \quad \mathit{pc}_2 : P_2}{\mathit{pc}_1||\mathit{pc}_2 : P_1 + P_2}$$

Figure 3: Typing rules $\mathit{pc} : P$ for pointcut compositions.

The types of `mset`, `sset`, `mget`, `mset` and `args` pointcuts are shown in Figure 1. For example, the type of the pointcut `mset(int Point.x)`, which matches `p.x = 3` assuming `p` is an instance object of a `Point` class, becomes $\{\texttt{target} : \texttt{Point}, \texttt{args} : \texttt{int}, \texttt{kind} : \texttt{mset}, \texttt{name} : \texttt{x}, \texttt{ret} : \bullet\}$.

The typing rules for pointcut compositions (i.e., $\mathit{pc}_1\&\&\mathit{pc}_2$ and $\mathit{pc}_1||\mathit{pc}_2$) are shown in Figure 3. Composing two pointcuts with an `or` pointcut, the resulting type becomes simply the union of the two pointcut types. Composing with an `and` pointcut, the resulting type becomes a common subtype of the two pointcut types, intuitively. The common subtype is calculated using the rules in Figure 4.

As we can see, we need to define the type subsumption ($<:$) on pointcut types only for the cases that the right hand side is a record type. It is simply defined as follows.

$$\bot <: \{l_i : {T_i}^{i\in 1\cdots n}\}$$

$$\frac{P_1 <: \{l_i : {T_i}^{i\in 1\cdots n}\} \quad P_2 <: \{l_i : {T_i}^{i\in 1\cdots n}\}}{P_1 + P_2 <: \{l_i : {T_i}^{i\in 1\cdots n}\}}$$

We employ the standard record type subsumptions (i.e. subsumptions on record widths, depths and permutations [18]).

$$\frac{n \le m \quad \forall i \in 1\cdots n.\exists j \in 1\cdots m.Y_i <: Y'_j}{\{l_i : {Y_i}^{i\in 1\cdots n}\} <: \{l'_i : {Y'_i}^{i\in 1\cdots m}\}}$$

For the elements of *idpats*, say T_1 and T_2, the subsumption is defined as follows. $T_1 <: T_2$ if

- T_2 is $*$, or
- T_1 and T_2 is the same identifier.

And the subsumptions on sequences of elements and tagged elements of *idpats* are defined as follows.

$$\frac{n = m \quad \forall i \in 1\cdots n.T_i <: T'_i}{[T{i}^{i\in 1\cdots n}] <: [{T'_i}^{i\in 1\cdots m}]}$$

$$\frac{g_1 = g_2 \quad T_1 <: T_2}{T_1^{g_1} <: T_2^{g_2}}$$

[1] Similar elaboration can be applied to pointcuts related to methods, i.e., `call` and `execution` pointcuts.

$$
\begin{array}{rcl}
\texttt{mget}(T\ C.f) & : & \{\texttt{target} : C, \texttt{args} : \bullet, \texttt{kind} : \texttt{mget}, \texttt{name} : f, \texttt{ret} : T\} \\
\texttt{sget}(T\ C.f) & : & \{\texttt{target} : C^{\bullet}, \texttt{args} : \bullet, \texttt{kind} : \texttt{sget}, \texttt{name} : f, \texttt{ret} : T\} \\
\texttt{mset}(T\ C.f) & : & \{\texttt{target} : C, \texttt{args} : [T], \texttt{kind} : \texttt{mset}, \texttt{name} : f, \texttt{ret} : \bullet\} \\
\texttt{sset}(T\ C.f) & : & \{\texttt{target} : C^{\bullet}, \texttt{args} : [T], \texttt{kind} : \texttt{sset}, \texttt{name} : f, \texttt{ret} : \bullet\} \\
\texttt{args}({T_i}^{i\in 1\cdots n}) & : & \{\texttt{args} : [{T_i}^{i\in 1\cdots n}]\}
\end{array}
$$

Figure 1: Typing rules $pc : P$ for `mset`, `sset`, `mget`, `mset` and `args` pointcuts ($T, C, f \in idpats$).

$$\bot \otimes P \rightsquigarrow \bot \qquad P \otimes \bot \rightsquigarrow \bot$$

$$\frac{P <: \{l_i : {T_i}^{i\in 1\cdots n}\} \quad P <: \{l'_i : {T'_i}^{i\in 1\cdots n}\}}{\{l_i : {T_i}^{i\in 1\cdots n}\} \otimes \{l'_i : {T'_i}^{i\in 1\cdots n}\} \rightsquigarrow P}$$

$$\frac{P_1 \otimes P_3 \rightsquigarrow P'_1 \quad P_2 \otimes P_3 \rightsquigarrow P'_2}{(P_1 + P_2) \otimes P_3 \rightsquigarrow P'_1 + P'_2}$$

$$\frac{P_1 \otimes P_2 \rightsquigarrow P'_2 \quad P_1 \otimes P_3 \rightsquigarrow P'_3}{P_1 \otimes (P_2 + P_3) \rightsquigarrow P'_2 + P'_3}$$

Figure 4: Type calculation rules $P \otimes P \rightsquigarrow P$

Note that `ArrayList` $<:$ `Object` is not available in our definition. This is mainly because we want to check pointcut compositions without any base programs.

The subsumption on kinds $k_1 <: k_2$ holds only when k_1 and k_2 are the same.

3.3 Typing Examples

This section demonstrates that our type system can successfully accept the well-formed pointcuts and detects never-matching pointcuts.

Never-matching pointcut compositions is typed as $\bot$.

Our type system can successfully type `get(* *)&&set(* *)` as $\bot$ without any base programs. For simplicity, we show this by using elaborated pointcuts, i.e., `mget(* *.*)&&mset(* *.*)`[2]. As shown in Table 1, each pointcuts are typed as follows.

`mset(* *.*)`:
$\{\texttt{target} : *, \texttt{args} : [*], \texttt{kind} : \texttt{mset}, \texttt{name} : *, \texttt{ret} : \bullet\}$

`mget(* *.*)`:
$\{\texttt{target} : *, \texttt{args} : \bullet, \texttt{kind} : \texttt{mget}, \texttt{name} : *, \texttt{ret} : *\}$

The type of the composed pointcut `mget(* *.*)&&mset(* *.*)` becomes a common subtype of the two pointcut types as mentioned in Section 3.2, and we find $\bot$, which is the only possible type because there is no common subtype of $\bullet$ and $*$, nor of `mget` and `mset`. Thus our type system types `mget(* *.*)&&mset(* *.*)` as $\bot$, and can conclude that it is a never-matching pointcut.

The union of never-matching pointcuts and well-formed pointcuts is not typed as $\bot$.

Composing a never-matching pointcut and a well-formed pointcut with an `or` pointcut (`||`), we get a well-formed pointcut. In our type system, the fact is rephrased that composing an pointcut typed as $\bot$ and another pointcut not typed as $\bot$ with an `or` pointcut, the type of the resulting pointcut is not $\bot$. This property is satisfied in our system clearly following to the typing rule for the `or` pointcut compositions.

For example, the pointcut

```
(mset(* *.*)||mget(* *.*))&&args(int)
```

is well-formed and matches all assignments to any object fields. Because get join points have no argument, none of them is selected.

Reducing the bottom types by using the assumptions $P + \bot = P$ and $\bot + P = P$, the type of the pointcut becomes

$$\{\texttt{target} : *, \texttt{args} : [*], \texttt{kind} : \texttt{mset}, \texttt{name} : *, \texttt{ret} : \bullet\}$$

in our type system and it successfully reflects the fact that we mentioned just before.

4. RELATED WORK

Our work is not the first attempt to detect never matching pointcuts. Douence et al. defined the alphabet analysis for their pointcut language for control-flow. An alphabet is a set of join point shadows [15] that can generate matching join points. They also suggested that when the alphabet becomes empty, the pointcut never matches any join points and such pointcut definitions are erroneous. Program Description Language (PDL) [16] is a domain specific language for checking design rules such as the Law of Demeter [14]. Its pointcut language is similar to the one of AspectJ, and has a type system that assures that typed pointcuts have at least one matching join point. The typing rule and semantics of `not` pointcuts are very interesting, although the semantics differs from the one of AspectJ.

Aspect FGJ (or shortly AFGJ) [11] is an aspect-oriented calculus which extends Featherweight GJ [10] with forms for advice declaration and for proceeding to the next declared advice. Though the language have a execution pointcut primitive `exe` and two operators for pointcut compositions `&&` and `||`, the pointcut logic can successfully reject pointcut compositions of two different execution pointcuts such as[3]

```
exe(int Point.getX())&&exe(int Point.getColor()).
```

We think this work may be a good starting point of our formalization task.

MiniMAO_1 [3] is another core aspect-oriented calculus of AspectJ-like aspect-oriented programming languages based on Classic Java [6] to investigate the semantics of proceed and the soundness over advice weavings. Types of pointcuts are similar to ours but the approach does not detect never-matching pointcuts.

[2]The `get` and `set` pointcuts shall be encoded by disjunctions of the `mget` and `sget`, and the `mset` and `sset` pointcuts, respectively.

[3]Though AFGJ has a different syntactic format like `exe int Point.X()`, we use AspectJ-like format for readability.

AspectML [19] is a polymorphic aspect-oriented functional programming language. Pointcuts are first-class values and typed, but the language merely has `execution` join points as far as we know.

5. CONCLUSIONS AND FUTURE WORK

We pointed out that the AspectJ family of languages allow compositions of pointcuts that never match join points in any program, and that such compositions should be detected from aspect definitions alone. We showed a sketch of our type system to detect such never-matching compositions of pointcuts. Our key idea is to encode types of pointcuts and join points with record types. In the type system, the type of mutually exclusive pointcut compositions, such as `set(* *)&&get(* *)`, becomes $\bot$, which denotes never matching pointcuts.

We are currently working on the details of the type system based on Featherweight Java [10]. One of the major difficulties we are facing now is the `!` (`not`) operator. One possible solution would be to use negation (or complement) types, whose semantics is based on sets [7].

Our future work includes proof of type soundness; i.e., for any non $\bot$-typed pointcuts there exits a join point that matches the pointcut. An interesting direction of our future work is to extend the languages with generics so that we can verify correctness of the design and implementation of pointcuts in AspectJ5.

Acknowledgments

We would like to thank the anonymous reviewers for their encouraging and thoughtful suggestions. We also thank Jan Hanneman, Kohei Sakurai, Kazunori Kawauchi, Tsutomu Kumazawa and other members of the TM seminar at the University of Tokyo, and Atsushi Igrashi and other members of the Kumiki project for their valuable advice and discussions on this study.

6. REFERENCES

[1] T. Aotani and H. Masuhara. Scope: an AspectJ compiler for supporting user-defined analysis-based pointcuts. In *Proceedings of the 6th International Conference on Aspect-oriented software development*, pages 161–172. ACM Press, 2007.

[2] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: An extensible AspectJ compiler. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, pages 87–98. ACM Press, 2005.

[3] C. Clifton and G. T. Leavens. MiniMAO$_1$: An imperative core language for studying aspect-oriented reasoning. *Science of Computer Programming*, 63(3):321–374, 2006.

[4] R. Douence and L. Teboul. A pointcut language for control-flow. In *Proceedings of 3rd ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, pages 95–114. Springer, 2004.

[5] M. Eichberg, M. Mezini, and K. Ostermann. Pointcuts as functional queries. In *Proceedings of the 2nd ASIAN Symposium on Programming Languages and Systems*, pages 366–381. Springer, 2004.

[6] M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer's reduction semantics for classes and mixins. In *Formal Syntax and Semantics of Java*, pages 241–269. Springer, 1999.

[7] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 137–146. IEEE Computer Society, 2002.

[8] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, 2000.

[9] K. Gybels and J. Brichau. Arranging language features for more robust pattern-based crosscuts. In *Proceedings of the 2nd international conference on Aspect-Oriented Software Development*, pages 60–69. ACM Press, 2003.

[10] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[11] R. Jagadeesan, A. Jeffrey, and J. Riely. Typed parametric polymorphism for aspects. *Science of Computer Programming*, 2006. to apper.

[12] A. Kellens, K. Mens, J. Brichau, and K. Gybels. Managing the evolution of aspect-oriented software with model-based pointcuts. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, pages 501–525, 2006.

[13] G. Kiczales, E. Hilsdale, J. Hugunin, et al. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[14] K. Lieberherr, D. H. Lorenz, and P. Wu. A case for statically executable advice: checking the law of demeter with AspectJ. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pages 40–49. ACM Press, 2003.

[15] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In *Proceedings of the 12th International Conference on Compiler Construction*, pages 46–60, 2003.

[16] C. Morgan, K. D. Volder, and E. Wohlstadter. A static aspect language for checking design rules. In *Proceedings of the 6th International Conference on Aspect-Oriented Software Development*, pages 63–72. ACM Press, 2007.

[17] K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, pages 214–240. Springer, 2005.

[18] B. C. Pierce. *Types and programming languages*. MIT Press, 2002.

[19] G. Washburn and S. Weirich. Good advice for type-directed programming aspect-oriented programming and extensible generic functions. In *Proceedings of the 2006 ACM SIGPLAN workshop on Generic programming*, pages 33–44. ACM Press, 2006.

On the relation of aspects and monads

Christian Hofer and Klaus Ostermann
Computer Science Department
Darmstadt University of Technology, Germany

ABSTRACT

The relation between aspects and monads is a recurring topic in discussions in the programming language community, although it has never been elaborated whether their resemblences are only superficial, and if not, where they are rooted. The aim of this paper is to contrast both mechanisms w.r.t. their capabilities and their effects on modularity, first by looking at monads as a way to express tangling concerns in functional programming and by discussing whether they can be regarded as a form of AOP, then by taking the view that monads express concerns of computations and by analyzing the extent to which aspects are able to handle those concerns.

Our results are mostly negative: monads are not capable of quantifying over points in the program execution in a declarative way, whereas aspects are not very useful in abstracting over computational capabilities.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*Languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features; F.3.3 [**Logic and Meanings of Programs**]: Studies of Program Constructs

General Terms

Design, Languages

Keywords

aspect-oriented programming, aspects, monads, monad transformers

1. INTRODUCTION

Since De Meuter [9] has first discovered resemblences between aspects and monads on the descriptive level – layering of code, system wide repercussions, easy integration – those are a recurring topic in discussions in the programming language community, although it has never been elaborated whether they are only superficial, and if not, where they are rooted. This paper tries to shed some light on this topic by contrasting the two mechanisms w.r.t. to their capabilities and their effects on modularity.

We first specify what we regard as the essence of each of the concepts. Then, in the next section, we will look at monads as a way to express tangling concerns in functional programming, and discuss the extent to which monadic programming can be regarded as a form of AOP. We therefore analyze a monads-based implementation of the display update example as used by Kiczales and Mezini [5] (and many others).

Finally, we take the view that monads express concerns of computations and we analyze the extent to which aspects are able to handle those concerns. This discussion is mainly theoretical, however a monadic interpreter (see Liang et al. [6]) for a small functional language with dynamic variable binding will be simulated in AspectJ to demonstrate the use of dynamic quantification.

Although this work started as a project to identify the commonalities of aspects and monads, it turned out that the differences prevail. Based on ones point of view, this may or may not be very surprising, but since the topic is repeatedly brought up in blogs or discussions, we believe there is some value in substantiating the debate.

In the following, we assume basic familiarity with monads and monad transformers as available in Haskell [11], and familiarity with AspectJ [3].

1.1 Aspect-oriented programming

The aim of AOP is generally uncontested: the separation of cross-cutting concerns, i.e. a better source code organization that prevents that concerns are *scattered* around the source code, and complementarily, that several concerns are *tangled* at single places.

It is less clear, how to characterize the actual mechanisms for achieving this aim. Filman/Friedman [1] have given the famous characterization of AOP as "quantification and obliviousness", meaning that it allows the triggering of actions whenever a specified condition arises in a program (quantification), without the knowledge of the programmer (obliviousness). But as obliviousness is in conflict with the principle of explicit interfaces, and there is a case for non-oblivious quantification as well, this definition has been seen as too restrictive. On the other hand, the definition of Masuhara/Kiczales, requiring "a common frame of reference that two (or more) programs can use to connect with each other and each provide their semantic contribu-

Sixth International Workshop on Foundations of Aspect-Oriented Languages (FOAL 2007), March 13, 2007, Vancouver, BC, Canada.

tion" [8] seems too general to delimit AOP from other kinds of module systems.

Another characteristic of current AOP mechanisms, the "introduction of declarative policy languages" [12], is left implicit in both definitions, but will turn out to be helpful in drawing a line to "classical" module systems.

For the further analysis, we will pragmatically define a mechanism as aspect-oriented, if it aims at the separation of concerns in the organization of source code, by making use of declarative quantification and by finding a balance between the contradictory principles of obliviousness and of explicit interfaces.

1.2 Monads

Monads, besides being a notion of category theory and a means of defining a categorical semantics of computations [10], are a mechanism in functional programming that (1) allows the introduction of imperative statements, like stateful computations or exceptions, into purely functional languages, and (2) provides a way to abstract over different kinds of computations. We want to recapitulate briefly, how this abstraction works.

A computation can be regarded as a kind of function that will typically produce a value. Each kind of computation is characterized by a specific structure of parameters and return values. It can be defined by a type constructor that – together with the value typically produced by a computation – defines the type of this computation.

For example, the type constructor `Maybe` defines computations that typically produce values, but may fail. The types of this kind of computation can be defined polymorphically as:

```
data Maybe a = Just a | Nothing
```

There are two important operations that go with every kind of computation: (1) several computations of the same kind can be put together into a sequence; (2) a value can be injected into a computation with the effect that this value will be returned by the computation when it is run. Haskell provides a specific monadic (`do`) notation that is useful for putting together computations. Let us consider a simple example of a comparison of two values associated with two keys in a database:

```
eqVal key1 key2 db =
   do val1 <- lookup key1 db
      val2 <- lookup key2 db
      return (val1 == val2)
```

This is a sequence of three computation steps, the first will lookup `key1` and will typically produce a value `val1`, but may fail. The second step works analagously. The third computation compares both values: the boolean value that results from the comparison is injected into the kind of computation. For the `Maybe` monad, the `return` operator is the `Just` constructor, so if the computation arrives at the third computation step, the result is either `Just True` or `Just False`. The sequencing of computations is left implicit in the `do` notation. A `bind` operator binds the value returned by the first computation to a variable that can be accessed by the latter computations. If a computation step fails in the `Maybe` monad, the complete sequence is aborted, and `Nothing` is returned.

The monad abstracts over the possible failure of the computation. The programmer is oblivious to what happens behind the scenes in two regards: (1) she does not have to worry about the specific structure of parameters and return values, but can simply inject values into the computation where needed; (2) she does not have to worry about how the effects of one step are passed to the next computation steps (in this case, this concerns only the abortion of the complete computation; but e.g. in the `State` monad, the passing of the state through all computations is hidden behind the scenes.)

Different monads can be combined via monad transformers in order to express more complex kinds of computations. The different concerns of these computations are separated into the individual monads.

Modularity effects.

The addition of computational capabilities to some operations is the cause of a severe modularity problem in functional programming: often the parameter structure of a whole set of functions within a program has to be adapted to reflect this additional capability. Using monads, the parameter structure can be encapsulated together with the operations that make use of it (in the case of the `Maybe` monad, the only such operation is the `fail` operation that is associated with the `Nothing` constructor). In that way, monadic programming allows the careful inclusion of specific computational powers into a program, while keeping the different kinds of computations modularized.

The price to pay for this is referential transparency, if one takes the `do` notation of Haskell literally. For example, the result of the `get` operation in the state monad is dependent on the context, although it appears not to take any input parameters. Of course, if one takes the hidden `bind` operator into accout, it is still valid to reason with value substitution.

2. TANGLING CONCERNS IN FP

The tangling of concerns is not restricted to imperative programming, but prevalent in functional programming as well. Its simplest form is the *side effect*. But tangling concerns can go together in different ways. A transaction concern e.g. could rewind a computation. Monads are a natural starting point, if one implements those tangled concerns in a purely functional programming language. They allow for the production of side effects as well as for some control of execution, hiding all those computational details from the base functionality.

In the following section, we want to present an implementation of the display update example in Haskell using monadic programming. We want to discuss the capabilities of monads regarding the modularization of the cross-cutting concerns inherent in this example. The core of the display update example – as it is discussed e.g. in [4] – is a module that defines two simple shapes, points and lines, on a two-dimensional cartesian coordinate system. All shapes are updateable structures. In particular, they all have a `move` operator, that moves the shape along both coordinates. This module is complemented by an aspect that is responsible for performing a display refresh, whenever a shape on the display is updated.

The whole example rests on an imperative programming foundation, regarding the shapes as stateful objects. It shall not be discussed here, whether this implementation is the

most natural for a functional programming language. As the programming style is monadic, anyway, the imperativeness of the task is no hindrance to the implementation and can still show the intricacies involved in handling tangling concerns.

Display module.

We do not want to discuss the display module in detail, but we cannot omit it completely: in contrast to the Java solution we have to be much more specific on which computational powers it shall have. The display functions are implemented via an IOable monad (i.e. a monad that implements a `liftIO` function – belonging to the `MonadIO` type class – that lifts an IO operation to the monad). Using the style of [2], we define the interface of the display functions via a type class, while giving an implementation using a state monad transformer.[1]

```
class MonadIO m => MonadDisplay m where
   setDisplay :: DisplayObject -> m ()
   refreshDisplay :: m ()

type DisplayT = StateT DisplayObject

instance MonadIO m => MonadDisplay (DisplayT m)
   ...
```

The `setDisplay` function can be used for specifying an object to be displayed, the `refreshDisplay` function has to be called, whenever some information on the display changed so that the information has to be refreshed.

An object that shall be displayed must be made an instance of the `Displayable` class and implement the `display` function.

```
class Displayable a where
    display :: MonadIO m => a -> m ()

data DisplayObject =
   forall a. Displayable a => DisplayObject a
```

The use of the `forall` quantifier in the data type `DisplayObject` can be regarded as a trick to ensure polymorphism over all `Displayable` objects in the `setDisplay` function.

Shapes module.

The basic shapes functionality is implemented imperatively, in the example. `IORef`s are used as references to memory cells: a point is a reference to a pair of integers, a line is a reference to a pair of points. Due to the use of `IORef`s, all computations have to take place in an IOable monad. The constructor `newPoint` and the accessor `getPointX` are examples of functions that do not perform a state change and solely depend on the `MonadIO` class.

```
newtype Point = P (IORef (Int, Int))

newPoint :: MonadIO m => Int -> Int -> m Point
newPoint x y =
    do p <- liftIO $ newIORef (x, y)
       return (P p)

getPointX :: MonadIO m => Point -> m Int
getPointX (P p) =
    do (x,_) <- liftIO $ readIORef p
       return x
```

In contrast, the `movePointBy` function has to trigger a display refresh. The straightforward way to integrate the shapes functionality with the display functionality is to import the `Display` module and add a call to `refreshDisplay` at the end of all operations that perform a state change. The `movePointBy` function then looks like this:

```
movePointBy (P p) dx dy =
    do liftIO $ modifyIORef p
                  (\(x, y) -> (x+dx, y+dy))
       refreshDisplay
```

The type signature for this function is inferred as:

```
movePointBy :: (MonadDisplay m) =>
               Point -> Int -> Int -> m ()
```

Analogously, all the other state modifying functions have to run in a monad encompassing the display state. For making points and lines displayable, they have to be declared instances of the class `Displayable`.

2.1 Obliviousness

This solution is very similar to a typical object-oriented implementation (see e.g. the "GOFP" solution in [5]). The base functionality is tangled with calls to the display module. The monadic style allows us to abstract from the current display state that would otherwise have to be passed around through the base code. But it does not allow us to separate out the call triggering the display refresh.

In AspectJ we can externally define a pointcut that triggers the display refresh and that the programmer of the base functionality is oblivious of. We cannot achieve obliviousness with monads. The module boundaries are clearly respected, and there is no way to reflect over the names of the computations. The situation would be different, if monads were used to implement an interpreter for an AO language, because inside an interpreter reflective access is possible. Indeed, Wand et al. [13] have defined a monadic semantics of a pointcut mechanism: each procedure call takes place within a join-point environment that is extended by the name of the currently called procedure. This access to the procedure name is not available in a direct implementation. Adding reflection to a language with monads, on the other hand, runs the risk of breaking not only modularity, but as well the monad laws.

2.2 Non-oblivious AOP?

As it has been argued that obliviousness is not essential for AOP, we can try to achieve some form of non-oblivious separation of concerns. We first have to specify, which concerns are involved in the example. We will follow Kiczales/Mezini who have analyzed three further concerns besides the refresh implementation and the base functionality:

"**Context-to-Refresh** – What context from the actual display state change points should be available to the refresh implementation?

"**When-to-Refresh** – When should the display be refreshed?

[1]The full code is available at `http://www.st.informatik.tu-darmstadt.de:8080/~ostermann/foal07/`

"**What-Constitutes-Change** – What operations change the state that affects how shapes look on the display, i.e. their position?" [5]

The separation of the "context to refresh" concern makes use of obliviousness in the implementations discussed in [5]. A static pointcut like `this`, `target` or `args` has to be used within the aspect module in order to query the relevant information from a single place. The alternative would be to pass along the information with the `refreshDisplay` call. That would leave the concern scattered within the base functionality. While a reader monad could avoid the explicit parameter passing, the environment to be supplied to the display module has still to be defined somewhere in the state change operations of the base code. Thus, the scattering cannot be avoided in that way.

The distinction between the concerns "when to refresh" and "what constitutes change" can be made in the monadic program, and the former concern can thereby be modularized. For this purpose, the call of `refreshDisplay` is omitted and the state changing code is embedded into the call of a function `withStateChangeSignal`, instead.

```
movePointBy (P p) dx dy = withStateChangeSignal $
    liftIO $ modifyIORef p (\(x, y) -> (x+dx, y+dy))
```

The `withStateChangeSignal` takes a computation as parameter, and embeds the computation into the sending of a signal.[2] This allows the receiver of the signal to attach other computations before, after or around (including: instead of) the original computation. This is in contrast to firing an event, which would be the classical OO solution.[3]

The signal is declared via a type class `MonadStateChange`, which in effect globalizes the declaration and permits to define an implementation in some module that is not imported by the shapes module.

```
class MonadIO m => MonadStateChange m where
    withStateChangeSignal :: m a -> m a
```

The type of the `movePointBy` function is inferred as:

```
movePointBy :: (MonadStateChange m) =>
               Point -> Int -> Int -> m ()
```

We leave the concern of "what constitutes change" tangled with the base functionality. We could separate the latter by factoring out the former into a module working as a proxy, that simply passes on the operations while accompanying those operations that perform a state change with the corresponding signal. The viability of this approach depends on the power to redirect to the proxy the calls to the shapes module by its clients, that – if it is to be done in a non-oblivious way – requires a powerful module system. To which extent the Haskell type class system is apt for this task, cannot be discussed here.

But even then, the tangling of concerns would only be shifted to the proxy. Tangling is inevitable, because in monadic programming there is no mechanism for what Filman/Friedman [1] call "static quantification": we cannot make quantified statements over the program text, in the sense of making statements that have an effect on more than one place in the elaborated program (see [1]).

[2]The `with` prefix is adapted from a Lisp macro convention.
[3]However a similar effect could be achieved in Java by encapsulating the state modifying code into an anonymous class.

2.3 Declarativeness

A separate module is responsible for the integration of the shapes and the display modules. It defines the `Displayable` instances for the shapes, and it implements the "when to refresh?" concern by defining an instance of the `MonadStateChange` class:

```
instance MonadDisplay m =>
         MonadStateChange m where
    withStateChangeSignal c =
        do result <- c
           refreshDisplay
           return result
```

The implementation evaluates the computation that has been provided and refreshs the display thereafter. However, in contrast to an AspectJ implementation (that depends on a `displayStateChange()` pointcut; adapted from [5]):

```
after() returning: displayStateChange() {
   Display.refresh();
}
```

it is obvious that the definition of when the display shall be refreshed is not done in a declarative manner.

2.4 Dynamic quantification

However, it is possible to implement what Filman/Friedman call "dynamic quantification", the tying of "aspect behavior to something that happens at run-time" [1]. In the current implementation, the `moveLineBy` code looks like this:

```
moveLineBy :: MonadStateChange m =>
              Line -> Int -> Int -> m ()
moveLineBy (L l) dx dy = withStateChangeSignal $
    do (p1, p2) <- liftIO $ readIORef l
       movePointBy p1 dx dy
       movePointBy p2 dx dy
```

As `movePointBy` signals a state change as well, the signal is sent three times. We can, however, adapt our implementation of the `withStateChangeSignal` function, in order to prevent a repeated display refresh:

```
type StateChangeT = ReaderT Bool

instance MonadDisplay m =>
 MonadStateChange (StateChangeT m) where
    withStateChangeSignal c =
        do result <- local (\_ -> True) c
           p <- ask
           unless p (lift refreshDisplay)
           return result
```

The monad is adapted by transforming the display monad through a reader monad over a boolean flag. The flag signals whether a need to refresh the display has already been registered by a surrounding computation. The `withStateChange` computation first executes the computation that has been provided as its parameter and keeps the result. This execution is embedded into an environment where the flag set, because the function itself takes responsibility for the display refresh. Afterwards, the computation will check its own environment, whether the flag is set, and trigger a display refresh otherwise. In any case, the kept result is returned.

The instance declaration ensures that the display monad transformer and the reader monad transformer of the integrating module are combined. The order of combination is irrelevant when combining a state and a reader monad transformer, but the implementation could potentially break if we were using another implementation of the display monad that would not just encapsulate a state monad.

When looking at the examples that Filman/Friedman [1] give for dynamic quantification, it is apparent, that some of them correspond directly to monads: raising of exceptions (error monad), calling a subprogram in temporal scope of another operation (reader monad), the history of the program execution (state monad). Furthermore, the authors note that AOP variants of other programming languages may include other ways of dynamic quantification, due to their native language features, and name the capturing of the current continuation in Scheme as an example (continuation monad).

On the other hand, monadic programming only allows non-oblivious dynamic quantification, i.e. quantification over properties that are captured explicitly by a monadic operation (that works in that regard as a semantic marker), while the typical use of e.g. the `cflowbelow` pointcut descriptor is quantification over the control flow based on syntactic names.

In addition, the monadic solution to the redundant display refresh problem is not declarative. It uses a sequential style for implementing the concern.

2.5 Advice confinement

Monads allow for a controlled extension of computational capabilities. Therefore we expect the handling of tangling concerns to be more controlled than in the AspectJ solution. Indeed, the Haskell type system gives us some guarantees on what part of the program the advice may affect. We know e.g. that the display refresh code cannot trigger a state change signal by some operation that it calls, because it does not run in an instance of the `MonadStateChange` monad. While this confinement of the powers of advice can simplify reasoning about the program, it may on the other hand be regarded as an unwanted restriction on the programmer's flexibility.

2.6 Conclusion

Monads are a common way to handle tangling concerns in purely functional programming. They are a traditional way to modularize computations in that they respect the module interfaces. They can therefore not achieve *oblivious* quantification. Furthermore, they differ from AOP by not allowing for *declarative* quantification. However they are similar to (a certain type of) AOP and more powerful than traditional module systems in one regard: they provide the abstractions that characterize *dynamic* quantification.

They appear to be similar to annotation-based AOP in that they are more powerful than a traditional modular solution, while remaining non-oblivious. Two differences to the annotation-based AspectJ solution are apparent, however: (1) the monadic solution does not allow for declarative quantification; (2) the AspectJ solution still encompasses a reflection mechanism via the `target`, `this`, and `args` pointcuts that break into the module implementation and allow for separation of the "context to refresh" concern.

3. ASPECTS OF COMPUTATIONS

While above monads were used to encapsulate specific tangling concerns, it can be argued that every monad can be regarded as expressing a concern: the kind of a computation can be regarded as an aspect of the computation. Based on this assumption, we want to analyze, to what extent aspects might be able to fulfill the role of monads in abstracting over kinds of computations. But we also want to shortly discuss, if the power of AOP to separate concerns were useful in monadic programming.

3.1 Abstracting over kinds of computations

At the heart of monads lie the two fundamental operators: the *return* operator that injects values into computations and allows the programmer to abstract over the parameter structure of the actual monad, and the *bind* operator that organizes the sequencing of computations. There are no equivalents for those operators in AOP. AOP is not about redefining the way that the sequencing of operations is interpreted. Instead it is about introducing additional action at specific points in the course of execution. This imposes severe limitations in the way that AOP mechanisms can manipulate the flow of control and enrich the parameter structure.

3.1.1 Manipulating the control flow

In AspectJ, after advice has been executed at some join-point, the execution is resumed after the join-point. There is no way to jump forward to some join-point matching another pointcut, or up to some position in the call stack. The only way to achieve the latter is by throwing an exception within the advice code, and by adding exception catching advice at the position where execution shall continue. But throwing of exceptions is not a mechanism introduced by AspectJ, but belongs to the mechanisms of the base language.

Generally, aspect languages seem not to provide mechanisms that allow the programmer to explicitly manipulate the control flow. Thus, we cannot hope to express computations that are able to fail, like those represented by the `Maybe`, the `Error`, or the `List` monad, via AO mechanisms. We will have to use exception mechanisms to jump out of the current point in program execution. The exception mechanism makes equal the very different kinds of computations expressed via the different monads, and therefore can hardly count as a good abstraction mechanism over them. W.r.t. the `Continuation` monad, if one does not restrict oneself to escape continuations, the situation is even worse, as the exception mechanism will probably not be powerful enough.

3.1.2 Enriching the parameter structure

An important part of abstracting over kinds of computation is associated with the ability to abstract over the parameter structure of a computation. As we do not have the power to inject values into computations, or to pass the hidden parameters along in AOP, we cannot expect to have a general mechanism for this kind of abstraction. The sequencing of computations can only be translated into a sequence of programming statements in our base language.

On the other hand, some of the powers offered by monads might already be included as part of our base language. For example, in Java there is no need to separate out the passing around of a state through the execution sequence, in the way it is done by the `State` monad. It might be argued

that AOP mechanisms allow the programmer to pass along state by introducing it within a separate module, e.g. by the inter-type declarations of AspectJ. But this is a redundant mechanism for abstracting over state, and is only useful for separating out a concern that is related to the state.

However there is a way to implement some hidden form of parameter passing in some AOP mechanisms, by using a combination of *dynamic* quantification and reflection.

3.1.3 Using dynamic quantification

The most prominent mechanism for dynamic quantification is the `cflow` pointcut in AspectJ. It allows for quantification over the control context and may be able to simulate powers of the `Reader` monad. We want to focus the discussion on this mechanism.

The reader monad is typically associated with a function to transform the environment of a control context (`local`) and a function to read the environment within a control context (`ask`). In order to simulate this in AspectJ, we have to define a pointcut that matches the point where the environment is transformed, and a pointcut that matches the point where the environment is demanded. Calling them `local()` and `ask()`, the pointcut for reading the environment can be defined as:

```
pointcut readEnvironment(Environment env):
   ask() && cflow(local(env));
```

The remaining problem is how to define the `local()` pointcut such that it contains the environment as a variable. First of all, it must be noted that due to the workings of the `cflow` pointcut, we can only access the innermost join-point that matches the `local()` pointcut in the context. Therefore, the complete environment must be made available there. We are able to collect the environment information by using one of the state-based pointcuts `this()`, `target()`, and `args()`.

A typical application of the reader monad is its use for keeping a variable environment in a programming language interpreter. Although it can be used for statically scoped variables, it is more naturally used for implementing (the less wide-spread) dynamic variable binding.

Let us look at a simple interpreter implementing dynamic binding for illustration purposes. A monadic interpreter written in Haskell could look like this:

```
data Term = Const Int
          | Var String
          | Lambda String Term
          | App Term Term

data Value = Num Int
           | Fun (Value -> Reader Environment Value)

interpret :: Term -> Reader Environment Value
interpret (Const c) = return (Num c)
interpret (Var varId) =
   do env <- ask
      return (lookupVar varId env)
interpret (Lambda varId body) = return $
   Fun $ \val -> local (\env -> (varId, val) : env)
                       (interpret body)
interpret (App e1 e2) =
   do Fun f <- interpret e1
      v <- interpret e2
      f v
```

In a non-monadic program, the environment would have to be passed through as a second parameter to the `interpret` function. The reader monad allows for increasing the modularity of interpreters by hiding this parameter (see: Liang et al. [6]).

In AspectJ, we can define a default implementation for a static method `Environment.read()` that returns an empty environment. This implementation is shadowed by an advice that is triggered, if the method is called within the context of an environment extension. The advice then returns the extended environment. The default implementation together with the advice plays the role of the `ask` function in the reader monad.

The interpreter for a `Var` term is implemented as follows (`Environment.lookup()` is a method that returns the value of a variable this is stored in the environment):[4]

```
public Value interpret() {
   return Environment.read().lookup(id);
}
```

The interpretation of a `Lambda` term is trivial: it creates a value of class `Function`, simply passing along its parameter name and its body:

```
public Value interpret() {
   return new Function(variable, body);
}
```

The `Function` class implements an `apply(Value)` method that will be called during the interpretation of an `App` term:

```
public Value apply(Value val) {
   return new InEnv(body,
                    Environment.read()
                        .extend(variable, val))
               .interpret();
}
```

The call of `local` in the Haskell code is replaced by the creation of a new term of a class `InEnv`, that is only meant to be used internally. The environment to be used for the function application is created by extending the surrounding environment via an `extend()` method on the `Environment` class. In order to get the environment that is to be extended, an `Environment.read()` message is sent here as well.

The `InEnv` class stores the function body as well as the environment in which the body shall be executed. Its `interpret()` method just calls the `interpret()` method of its body. Its sole aim is to store the environment such that it can be accessed by a pointcut. Whenever `Environment.read()` is called, the aspect code can access the environment in the following advice:

```
Environment around(InEnv inEnv):
   execution(* Environment.read())
   && cflow(execution(* InEnv.interpret())
            && target(inEnv)) {

   return inEnv.getEnvironment();
}
```

In this way, the `cflow` and `target` pointcuts can achieve the same effect as the `Reader` monad. The `local` function cannot be perfectly imitated for two reasons: firstly,

[4]The full code is available at `http://www.st.informatik.tu-darmstadt.de:8080/~ostermann/foal07/`

its first argument is an environment transformer, while due to the nature of the `cflow` pointcut, the complete environment must be available in the innermost join-point matching the `cflow` pointcut. And secondly, its second argument can be any function, while the `cflow` pointcut has to define the join-point, at which the environment is extended, by name. In the solution given above, these issues could be solved by making use of a new class `InEnv` that stores the environment, and by its method `InEnv.interpret()` that serves as a marker for the `cflow` pointcut.

3.2 Separating kinds of computations

In the following, it shall shortly be discussed whether the lack of separation of concerns when using monads has to be regarded as a shortcoming in relation to AOP. The question is, whether it is useful to separate the operations that access and manipulate the extended parameter structure from the normal execution of the sequence of operations.

In the example of the interpreter given above, the separation of the environment retrieval from the interpreter concern would be artificial: it is relevant for the understanding of how the interpretation of e.g. the `Var` term takes place to know that the environment is accessed at that point.

In contrast, there are situations in which the separation of concerns might seem appropriate: one could think of some logging function that is enabled or disabled by setting a dynamic boolean variable. It could be useful to put the logging code into a separate module.[5]

Something similar holds for the error monad. While Lippert/Lopez [7] give examples of the usefulness of separating out exception detection and handling into separate modules, this is not generally so. The basic reason for throwing an exception is that a computation cannot continue with a reasonable result. This breakdown in the execution of the current concern is normally a relevant part of this concern and it therefore is not appropriate for singling it out into a separate module. Looking at these two examples, it has to be expected, that there is no general answer to this question.

3.3 Conclusion

There is no general way to introduce computational powers in a controlled fashion in AOP. State and exceptions are part of the native mechansims of most languages. Their use cannot be restricted. Nevertheless, a certain extension of computational powers can be achieved by the mechanisms of dynamic quantification.

The limitation of referential transparency that has been discussed as a problem of the use of monads is omnipresent in those languages, anyway. On the other hand, the absent power of monads to separate out concerns, in the way that AOP does it, can in some cases be regarded as a limitation of their expressiveness.

4. CONCLUSION

To sum it up, monads and aspects have to be regarded as quite different mechanisms, not able to express each other. On the one hand, monads are not capable of oblivious quantification. The only kind of quantification they allow is dynamic quantification in a non-declarative way. It would therefore stretch the meaning of AOP to still consider monads as an aspect-oriented mechanism. On the other hand, aspects are not very useful in abstracting over computational capabilities.

[5]Of course, logging in Haskell requires the combination with some writer or similar monad.

5. REFERENCES

[1] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. Technical report, 2000.

[2] M. P. Jones. Functional programming with overloading and higher-order polymorphism. In *First International Spring School on Advanced Functional Programming Techniques*, number Lecture Notes in Computer Science 925, 1995.

[3] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[4] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 49–58, 2005.

[5] G. Kiczales and M. Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In *ECOOP*, pages 195–213, 2005.

[6] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–343, New York, NY, USA, 1995. ACM Press.

[7] M. Lippert and C. V. Lopes. A study on exception detection and handling using aspect-oriented programming. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 418–427, New York, NY, USA, 2000. ACM Press.

[8] H. Masuhara and G. Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *ECOOP2003*, July 2003.

[9] W. D. Meuter. Monads as a theoretical foundation for aop. position paper in ecoop '97 workshop on aspect-oriented programming.

[10] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[11] S. Peyton Jones. *Haskell 98 Language and Libraries: the Revised Report*. Cambridge University Press, 2003.

[12] M. Wand. Understanding aspects: extended abstract. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 299–300, New York, NY, USA, 2003. ACM Press.

[13] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Trans. Program. Lang. Syst.*, 26(5):890–910, 2004.

On bytecode slicing and AspectJ interferences

Antonio Castaldo D'Ursi
DEI, Politecnico di Milano
Via Ponzio 32
Milano, Italy
ercasta@gmail.com

Luca Cavallaro
DEI, Politecnico di Milano
Via Ponzio 32
Milano, Italy
cavallaro@elet.polimi.it

Mattia Monga
DICo
Università degli Studi di Milano
Via Comelico 39
20135 Milano, Italy
mattia.monga@unimi.it

ABSTRACT

AspectJ aims at managing tangled concerns in Java systems. Crosscutting aspect definitions are woven into the Java bytecode at compile-time. Whether the better modularization introduced by aspects is real or just apparent remains unclear. While aspect separation may be useful to focus the programmer's attention on a specific concern, the oblivious nature of the weaving makes it difficult to figure out the behavior of the whole system. In particular, it is not easy to figure out if two aspects interfere one with the other. We built a bytecode slicer called XCutter in order to study which part of the woven code is affected by the application of an aspect. However, our experiments show that a static analysis of AspectJ woven bytecode does not give the expected results, unless the code is properly annotated.

Categories and Subject Descriptors

D.1.m [**Programming Techniques**]: Miscellaneous; D.2.4 [**Software Engineering**]: Software/Program Verification; D.3.3 [**Language Constructs and Features**]:

General Terms

Languages, Verification

Keywords

Program analysis, Slicing, aspect-oriented programming, AspectJ, interference analysis

1. INTRODUCTION

AspectJ [1] is the most successful language embodying the idea of aspect-oriented programming, introduced by Kiczales et al. in [2]. In AspectJ, crosscutting entities called *aspects* are *woven* into traditional object-oriented (Java) bytecode at compile-time. Nevertheless, events that can trigger the execution of aspect-oriented code are run-time events: method calls, exception handling, and other specific points in the control flow of a program. The basic idea is that aspects describe crosscutting computations (pieces of *advice*) by referring to an abstract view of the system and composition is performed by the automatic weaving process, which produces a standard Java bytecode application.

In principle, the various aspects should not interfere with one another, and they should not interfere with the evolution of classes. Currently, non interference in presence of class evolution is really hard, and programmers should be very careful in writing aspects that make use of the implementation details of classes as little as possible if they want to be able to reuse their aspects. Moreover, it is still not clear how to cope with the difficult problem of aspect interaction. In fact, the code affected by an aspect is oblivious about that, i.e., its text does not contain any clue about which aspects might or will be advised on it. Thus, by looking at a given statement, programmers may have a hard time figuring out if one of the aspects of the system will influence it. We believe that this represents a limit in the current AspectJ approach, since it means that the actual separation of aspects is in a sense only apparent. In other words, while aspect code units are physically separated, one has always to keep all of them in mind while coding the other parts of the system, since every aspect could potentially influence any other component.

In order to assess the complexity of the weaving in an AspectJ program, we proposed[3, 4] to use program analysis techniques to measure how large is the portion of a program potentially (i.e., statically known) affected by an aspect. We suggested this could be used to study aspect interactions. In fact, roughly speaking, if the portions affected by two aspects do not overlap, this is a sufficient condition to state that they do not interfere. More precisely, let a *code unit* be an aspect or a class of a system. We say that an aspect A does not interfere with a code unit C if and only if every interesting predicate on the state manipulated by C is not changed by the application of A. For example, if an object x manipulated by C exists such that the predicate $x \leq 0$ must hold for the correctness of the system, A does not interfere with C only if C woven with A preserves $x \leq 0$. This definition captures only interferences caused by inconsistencies in the state manipulated by the code units: other types of clashing are not considered.

This can be used to derive an operational test to find out aspect potential collisions [3]. If A_1 and A_2 are two aspects and S_1 and S_2 the corresponding backward slices [5] obtained by using all the statements defined in A_1 and A_2 as the slicing

Sixth International Workshop on Foundations of Aspect-Oriented Languages (FOAL 2007), March 13, 2007, Vancouver, BC, Canada.

criterion, A_1 does not interfere with A_2 if $A_1 \cap S_2 = \emptyset$. In fact, the set S_2 contains all the statements that affect the slicing criterion (a set containing all the statements of A_2).

To this end, we built XCutter [1] (to be pronounced *crosscutter*) a tool to do backward static slicing on Java bytecode and to map bytecode entities back to the AspectJ code. In this paper we report about the challenges we encountered in building such a tool and some preliminary results about its use. The paper is organized as follows: Section 2 briefly surveys slicing techniques and talks about our work to build a working slicer, Section 3 talks about the preliminary results we obtained using our tool on AspectJ programs for interference detection, and Section 4 finally discuss the lessons we learnt.

2. SLICING OBJECT AND ASPECT ORIENTED PROGRAMS

Program slicing is a program analysis technique introduced by Weiser in the '80s [5]. A backward slice is a set of program instructions that, fixed one or more instructions as a *slicing criterion*, influence the criterion, i.e., change the right-values of the statements included in the criterion. Ottenstein and Ottenstein proposed a slicing technique based on a graph representation of the program in [6], in which directed edges represent data and control dependencies between instructions. The original proposal was about a technique to analyze a single procedure of a procedural program. In [7] the graph-based approach was extended to handle programs including interacting procedures calls, and an algorithm that could correctly take the calling context into account was proposed. This algorithm performs two phases of reachability analysis, which consider different kinds of edges, to preserve the calling context. The resulting slice is a set of graph nodes that is mapped back onto the program source. Slicing techniques were further studied and applied to object oriented programs by Liang and Harrolds in [8].

Object oriented slicing techniques, unfortunately, cannot be used as is to analyze aspect oriented programs, since the weaving introduces data dependencies that have to be taken into account. A graph representation for the AspectJ language was proposed by Zhao in [9]. His work relies on the graph representation presented for object oriented programs and adds representation for some of the constructs of AspectJ. Pieces of advice are represented as methods, pointcuts are represented adding a *pointcut edge* from the entry of a piece of advice representation to the point in the base system code captured by the pointcut. Intertype declarations are represented adding the representation for the field or method introduced and binding it with an *introduction edge* to the interested point of the base system. However, the dynamic nature of pointcut definitions (that can even apply to pieces of advice to which are attached) is neglected.

Slicing AspectJ programs by considering aspects as first class entities, is appealing, since it allows for not considering the actual implementation of the weaver. However, in order to build working tools one has to deal with the details of the expressive power of all AspectJ constructs. This effort actually replicates the weaving task. Therefore, in [4] we proposed to exploit the fact that AspectJ programs are eventually translated into Java bytecode, and the latter is an object oriented language, to which the mainstream state of the art of program analysis can be applied.

Our strategy is divided in four steps:

1. Compile Java classes and aspects using an AspectJ compiler and weave aspects into an executable program.
2. Apply existing slicing algorithms to the resulting bytecode.
3. Obtain a slice as a set of bytecode statements.
4. Map these statements back onto the original source code of the program.

Working at bytecode level may seem inappropriate, since in bytecode there is no distinction between the aspect oriented and the object oriented parts of an AspectJ program. In fact, the weaving process translates aspects into classes, pieces of advice in methods and pointcuts into method invocations. Thanks to this, the mapping of bytecode instructions onto source code can be done rather efficiently and precisely. Some problems about intertype declarations remain (see Section 2.3.5): these could be in future resolved by a suitable use of bytecode annotations by the AspectJ compiler. XCutter, our backward static slicer, can analyze both AspectJ and Java programs, since it works at bytecode level.
A tool similar to ours is Indus [10], a slicer for Java that works at the bytecode level: unfortunately it was made available when our effort was already begun and initially released with a license [11] incompatible with our commitment to produce an open source product. Indus can slice multithreaded programs, while our current prototype can not. It is based on a context-sensitive points-to analysis, but context-sensitivity is not fully exploited by its slicing algorithm. For example, the context-sensitive points-to analysis can distinguish between different instances of internal data structures of different `Vector`s, but the slicing algorithm does not distinguish between modifications to two different `Vector`s, thus obtaining the same precision level resulting from the use of a context-insensitive points-to analysis. Moreover, Indus is focused on slicing Java programs. AspectJ-specific slicing strategies could not be implemented on top of it, so it could not be used for interference analysis.

2.1 Slicer Architecture

XCutter is built on top of Soot, a program analysis framework for Java [12]. Soot provides an intermediate representation of Java bytecode called Jimple– a three-address typed representation suitable for analysis– and supports intraprocedural analyses. Moreover it is accompanied by Spark, a framework for points-to analysis.

XCutter is structured as a series of analyses which run inside the Soot framework. As depicted in Figure 1, the analysis starts by compiling the AspectJ program sources and

[1]The source code of the tool is available at `http://www.elet.polimi.it/upload/cavallaro/thesis/Xcutter.jar` under the terms of a GPL license.

weaving the resulting bytecode. The latter is then imported in Soot and translated into Jimple. This intermediate representation undergoes a series of preliminary analyses, used to discover some information necessary to build a slice, and is used as input for the slicing algorithm, described in Section 2.3. The resulting slice is finally mapped back onto the source code of the AspectJ program.

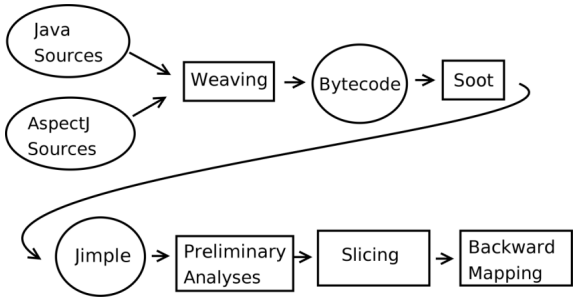

Figure 1: Architecture of XCutter

2.2 Preliminary analyses

Preliminary analyses compute data and control dependencies between instructions of the program. Instructions are annotated with information on the discovered dependencies, which are used by the slicing algorithm to compute the backward static slice.

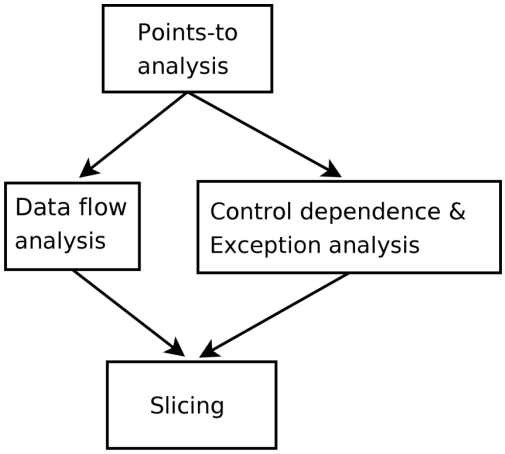

Figure 2: Overview of analysis order

The preliminary analysis starts computing the points-to information by using the Spark framework [13]. This analysis computes the set of Java objects that a given variable in the program may point to. These results are then used by both data flow and control dependence and exception analysis (see Figure 2).

The data flow analysis computes the reaching definitions in the program. For efficiency reasons we divided this analysis in local data flow analysis and reference data flow analysis. Local data flow analysis computes reaching definitions between Jimple local variables contained in a single method. For this analysis we adapted the algorithm described in [14]. Reference data flow analysis computes data dependencies caused by definitions and uses of static fields, object fields, and arrays. This analysis needs a side effects analysis as a preliminary step. The side effect analysis computes which object fields, arrays and static fields may be used or modified by each method in the program. Each method is, initially, analyzed by itself, then the information found for a method is propagated through the call graph. Reference analysis uses the results of the side effect analysis to model the effect of method call statements. Using this strategy the reference analysis can be performed intraprocedurally, improving efficiency.

The control dependence analysis aims at finding intraprocedural control dependencies. The algorithm used was adapted from [14]. The exception analysis was adapted from the one proposed in [15]. Exceptions may introduce intraprocedural and interprocedural control dependencies in a program. Methods in the program are searched, from call graph leaves to the main method, to find if they might throw exceptions. If an exception is thrown there is a control dependence from the throwing instruction to all the following instructions, in the method body. Moreover the method is searched for an appropriate catch clause. If it is found, it means there is a control dependence from the throwing instruction to the instructions contained in the catch block. If no catch block is found the information about the thrown exception is propagated to the callers. If the callers contain an appropriate catch block there is an interprocedural control dependence from the thrower instruction to the instructions of the catch block, else the information is further propagated to the callers.

The results of data flow and control dependence and exceptions analyses are annotated in tags associated with Jimple instructions and methods, and are used as input for the slicing algorithm.

2.3 The slicing algorithm

Existing slicing algorithms for procedural and object oriented programs ([7], [8], [16]) require the construction of a graph representing the analyzed program. Most features of the Java language have been separately taken into account, and algorithms to create corresponding graphs have been proposed. However, creating a graph that correctly takes into account the whole Java language requires theoretical work to merge different approaches. Our slicing algorithm is not graph-based. Instead of building a graph representing the entire program, slicing is performed using results of preliminary analyses, which compute dependencies between statements. This makes the slicing algorithm more complex than a graph-based algorithm, because dependencies between instructions are not represented explicitly by edges. However, several features such as exception handling and polymorphism are easier to manage without an unnecessary pollution of ad-hoc edges. In the following, we describe the engineering challenges we encountered in building a working tool: most of problems we faced are extensively studied in the program analysis literature. Notwithstanding that, putting together independent results in an effective prototype was a hard work, mostly absent in the publicly available code.

2.3.1 Library and application methods

Any non trivial Java program uses some library method. This means that a lot of library methods are part of the program, because they are transitively called by application code. Traditional algorithms require a detailed analysis of the entire program and the creation of a graph for every library and application method. While experimenting on small AspectJ examples, we noticed that usually a large part of the program is composed by library methods. Our interference analysis deals with slices containing instructions belonging to aspects, which are not contained in the Java library. So we decided to keep the distinction between library and application methods, and analyze them with different precision levels. In particular, detailed data and control dependence analysis is performed on application methods only. The slicer analyzes library methods to detect which values are used or defined by these methods, but it does not compute dependencies between instructions of a library method. This also affects how method calls are handled. Information on side effects and thrown exceptions is necessary to take into account data and control dependencies. However, calls to library methods are treated atomically. In fact, when an instruction calling a library method is put into the slice, the whole library method, and the method it transitively calls, are considered to be part of the slice. The difference between library methods and application methods is decided by the user, by choosing which packages contain application methods. Moreover, the analysis can be configured to treat some library methods with the same level of detail used for application methods, using a *depth* parameter. The increased precision is used for library methods whose distance from application methods in the call graph is smaller than *depth*.

2.3.2 Using dependencies

The slicing algorithm uses dependencies computed by preliminary analyses to add instructions to the slice. Although the slicing algorithm is not graph based, we called node the entity used to represent instructions. However, new nodes are created only when new instructions are added to the slice. The algorithm uses several kinds of nodes. When a new instruction is added to the slice, a new node is created, whose kind depends on the included instruction. Table 1 summarizes node types and the corresponding actions, and Figure 3 shows how the algorithm decides the type of a node when a new instruction is added to the slice.

Simple nodes are used to represent instructions not containing method calls, while *call site* nodes are used for instructions containing explicit or implicit[2] method calls.
Actual in and *actual out* nodes are used to represent values used or defined by a method at call sites. An actual in node represents a single value used by a method, such as a method parameter, an array, an object field or a static field. An actual out node represents a single value defined by a method.
Values used and defined by library methods are not represented using actual in and actual out nodes. A single *pseudo actual* node is used to represent all the values used and defined by a library method. In fact, since library methods are not analyzed in detail, there is no way to determine dependencies between output and input values. Using a single node to represent all of them provides a safe approximation, representing the fact that any output value could depend on any input value. Actual in nodes are only created when actual out nodes are examined.

[2]implicit method calls are calls to class static initializers

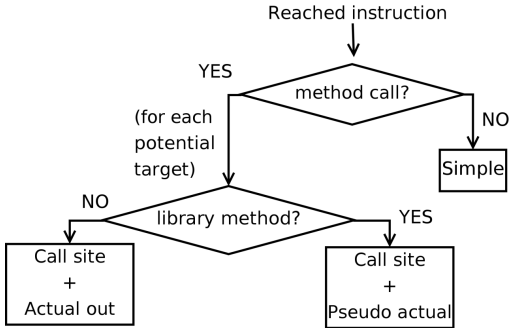

Figure 3: The flowchart for identifying pseudo-actual nodes

Nodes included in the slice are put in an *open list*. The algorithm extracts nodes from the open list one at a time, and executes different actions according to the node type (as shown in Table 1). For example, when a simple node is extracted from the open list, the algorithm examines data and control dependencies of the instruction represented by the node. Instructions on which the node depends are added to the slice, and corresponding nodes are added to the open list. However, when a call site node is extracted from the open list, control dependencies are examined as they are for simple nodes, but data dependencies are treated differently. In particular, the algorithm only examines data dependencies regarding the local variable on which the method is called. In fact, data dependencies regarding values used by the method are examined when the corresponding actual in nodes are examined. Once a node is examined, it is put in a *closed list*, which is used to avoid re-analyzing the same node.

2.3.3 Dependence relation

The most expensive part of the construction of the graph is the computation of summary edges (a detailed analysis of its cost is provided in [17]), that express the dependence of values defined by a method on values used by the same method. Graph-based slicing algorithms such as [7] require computing summary edges before the slicing phase begins. This can be very expensive in terms of required memory and computation time. For a typical Java program, the cost is $O(CallSites \times Params^3)$, where $Params$ is the maximum number of method parameters and $CallSites$ is the number of method call instructions in the code. Method parameters include object and static fields which are transitively used or modified by the method. Even for example programs using library methods, $Params$ is greater than 10,000 and $CallSites$ is greater than 100,000. This is why our algorithm computes and stores these dependencies during the slicing phase, using a dependence relation that is enriched as new dependencies are computed. When a method call instruction is put into the slice because of a data or control dependence, the computation of the dependence relation for

Node kind	Represents	Action
Simple	An instruction not containing a method call	Follow data and control dependencies
Call site	An instruction containing a method call	Follow control dependencies (and data dependencies for the local variable representing the receiving object)
Actual in	A value used by a method at a call site	Follow data dependencies for the value represented by the node
Actual out	A value defined by a method at a call site	Compute dependence relation, generate actual in nodes
Pseudo actual	All values used and defined by a library method at a call site	Follow data dependencies for every value used by the library method

Table 1: Actions corresponding to different kinds of slicing nodes

the related value is started. The value can be any value defined by the method, including thrown exceptions. To compute the dependence relation for a given value, the algorithm looks for the instructions that define the value. Then the algorithm starts examining dependencies according to table 1. During computation of the dependence relation, dependencies are used to reach other instructions. Values used by reached instructions are used to enrich the dependence relation. In fact, since the algorithm follows data and control dependencies, the values used by reached instructions influence the defined value. When the dependence relation is enriched, actual out nodes related to the examined method are analyzed again to create appropriate actual in nodes.

2.3.4 Current limitations

XCutter has currently some limitations. Some of them have effects on the correctness of the slice.

The Java language allows the programmer to call methods written in native languages, such as C and C++, using the Java Native Interface [18]. The slicing engine cannot analyze these methods, because there is no bytecode corresponding to them and thus Soot cannot create Jimple representations for them. Unfortunately, native methods might have side effects and not taking into account these side effects leads to incorrect slices. In the future we plan to add support for side effect specification of native methods.

Our slicer works under a closed world assumption. Some Java features do not respect this assumption, so our tool can not handle dynamic class loading and reflection, since they introduce in the program some elements unknown at compile time.

The slicing engine uses data and control dependencies to compute the slice. These two kinds of dependencies correctly describe sequential programs. To correctly take into account concurrent programs, however, other kinds of dependencies are needed. *Divergence dependencies*, *Interference dependencies*, *Synchronization dependencies*, and *Ready dependencies*, are used to model dependencies caused by synchronization and concurrency mechanisms [19]. The slicing engine does not consider these other kinds of dependencies, potentially and incorrectly excluding some instructions from the slice. However, data and control dependencies between instructions executed in the same thread are correctly taken into account.

2.3.5 Source code mapping

The computed slice is made of bytecode instructions, but it can be mapped back onto the source code, using source line information introduced by the weaver. Some instructions, however, are not correctly mapped. For example, most pointcut definitions are not mapped, because they generate no executable bytecode. In fact they are used by the weaver to identify join points where advice code has to be inserted. Another mapping problem is caused by *declare parents* or *introduce* instructions. These instructions are used by the weaver to modify the class hierarchy or the interface of the object oriented part of the program, but the weaver does not leave any trace of the modification in the bytecode, so these instructions are never included in the bytecode-level slice. To ease the work of bytecode analysis, we suggest that the weaver should put more information in the woven bytecode, exploiting, for example, the opportunity of annotating byte code introduced in Java5.

3. INTERFERENCE ANALYSIS

We exploit our slicer to study aspect interference. Consider the example shown in Listing 1.

The aspect `SpeedController` is interested in the calls to the "setters" of the `Factory` class: it regulates the speed, keeping it under a fixed value. The aspect `RotationMonitor` is in charge to log any speed change. While `SpeedController` modifies a property of the underlying system, the `RotationMonitor` is simply an observer. Thus, the `RotationMonitor` aspect does not interfere with the `SpeedController` one. (Conversely, the `SpeedController` does interfere with `RotationMonitor`).

We expected to be able to check this property with our slicer: a backward slice associated to `SpeedController` should not contain any of the statements of `RotationMonitor`. Unfortunately, things are more complicated. In fact, the dynamic nature of join-points selection means that the weaver has to put some machinery in the code. Modern weaver implementations use to translate each piece of advice as a method and to insert the translated code into the right point in the program, selecting a *Join point shadow* (i.e. the representation of a join point in the source code)[20]. They try, anyway, not to inline code to let the translated bytecode have the

Listing 1: The source code of two aspects to show interference definition

```
1  package examples.lollypop;
2
3  public aspect SpeedController {
4
5          pointcut speedset(Factory f,int x):
6                  call (public * Factory.set* (int) ) && args(x)
7                          && target(f);
8
9          after(Factory fact, int speed): speedset(fact,speed) {
10                 if (speed>4) {
11                         fact.setRotationSpeed(speed/2);
12                         System.out.println("check done");
13                 }
14         }
15 }
16
17
18 public aspect RotationMonitor {
19
20         pointcut speedmonitoring(Factory f,int speed):
21                 call (public void Factory.set* (int)) &&
22                 target(f) && args(speed);
23
24         after(Factory fact,int rpm): speedmonitoring(fact,rpm) {
25                 System.out.println("Lollypop stick rotation speed set to " + rpm + " rpms");
26         }
27 }
```

same accessibility rules than regular Java bytecode.
Following this approach it is sometimes necessary duplicating pieces of advice to translate properly a pointcut. An example of this behavior can be the *After Finally Advice*. This represents advice that should run after exiting from the selected join point, both in case of normal execution or in case of exception throwing. The translation strategy of the AspectJ compiler, in this case, is duplicating the call to the method that translates the given piece of advice. This implies the existence of a control dependence from the join point shadow to the advice methods call present in the normal execution branch and in the exceptional execution branch.
Moreover aspects are usually implemented following the singleton pattern (i.e. there is only one instance of each aspect in the system). The access to the aspect instance happens using the `aspectOf` static method of the aspect, that returns the required instance. This introduces a data dependence that is not present in the source code of the system.
An example of after finally advice translation is shown in listing 2. This listing shows the Jimple translation of the piece of advice of `SpeedController`. The statements at lines 26 and 42 are introduced by the translation of the after finally advice. To force the system to execute the piece of advice both in case of normal execution or in case of exception, at the end of the join-point shadow, is thrown an exception that is caught by instructions. in lines 57 and 58 introducing control dependencies that are not present in the source code of the system.
Lines 28 and 34 invokes the `aspectOf` method of `RotationMonitor`. This method returns an instance of the aspect itself. This is necessary since the piece of advice of `RotationMonitor` needs to execute after a speed change. The method `aspectOf` might throw a `NoAspectBoundException`. This exception can be caught at line 42, generating a control dependence from line 38 in listing 3 to the catch instruction at line 42 of listing 2.

These control dependencies, caused by the exception handling code introduced by the weaver, cause the interference analysis to assume that the two aspects interfere, even if, theoretically, we would expect no interference. Finding these dependencies is an important improvement in the accuracy of our prototype: our first version (described in [4]) could be successfully used to exclude interference between aspects like `RotationMonitor` and `SpeedController`, since it performed simpler, though potentially incorrect, analyses.
The spurious dependencies disappear if the pieces of advice shown in listing 1, which are of type `after finally`, are transformed into `after return` pieces of advice. In this case, the bytecode of the `SpeedController` aspect is simplified and does not use exceptions to manage control flow, as shown in listing 4.

There is no definitive solution to this problem since the dependencies are due to the semantics of the after finally advice. It should be, anyway, possible to ignore the dependency introduced by the translation of this kind of advice annotating, during the translation phase, the exceptions introduced. During the slicing phase the dependencies due to annotated exceptions can be ignored. This solution leaves unaltered the translated bytecode and does not alter the analysis semantics, since those exceptions, whose dependencies are ignored, are used only to transfer control.

4. LESSON LEARNED

Aspect oriented programming as popularized by AspectJ claims that cross-cutting concerns should be coded in iso-

Listing 2: The Jimple translation of the after advice of the aspect SpeedController in Listings 1

```
1  public class examples.lollypop.SpeedController extends java.lang.Object
2  {
3      public void ajc$after$examples_lollypop_SpeedController$1$fda05aef(examples.lollypop.Factory, int)
4      {
5          examples.lollypop.SpeedController r0, $r9, $r10;
6          examples.lollypop.Factory r1, r2;
7          int i0, i1;
8          java.lang.Throwable r3, r4, $r5, $r8;
9          examples.lollypop.RotationMonitor $r6, $r7;
10
11         r0 := @this: examples.lollypop.SpeedController;
12         r1 := @parameter0: examples.lollypop.Factory;
13         i0 := @parameter1: int;
14         if i0 <= 4 goto label7;
15
16         i1 = i0 / 2;
17         r2 = r1;
18
19     label0:
20         virtualinvoke r2.<examples.lollypop.Factory: void setRotationSpeed(int)>(i1);
21
22     label1:
23         goto label3;
24
25     label2:
26         $r5 := @caughtexception;
27         r3 = $r5;
28         $r6 = staticinvoke <examples.lollypop.RotationMonitor: examples.lollypop.RotationMonitor aspectOf()>();
29         virtualinvoke $r6.<examples.lollypop.RotationMonitor: void
30 ajc$after$examples_lollypop_RotationMonitor$1$839313f3(examples.lollypop.Factory,int)>(r2, i1);
31         throw r3;
32
33     label3:
34         $r7 = staticinvoke <examples.lollypop.RotationMonitor: examples.lollypop.RotationMonitor aspectOf()>();
35         virtualinvoke $r7.<examples.lollypop.RotationMonitor: void
36 ajc$after$examples_lollypop_RotationMonitor$1$839313f3(examples.lollypop.Factory,int)>(r2, i1);
37
38     label4:
39         goto label6;
40
41     label5:
42         $r8 := @caughtexception;
43         r4 = $r8;
44         $r9 = staticinvoke <examples.lollypop.SpeedController: examples.lollypop.SpeedController aspectOf()>();
45         virtualinvoke $r9.<examples.lollypop.SpeedController: void
46 ajc$after$examples_lollypop_SpeedController$1$fda05aef(examples.lollypop.Factory,int)>(r2, i1);
47         throw r4;
48
49     label6:
50         $r10 = staticinvoke <examples.lollypop.SpeedController: examples.lollypop.SpeedController aspectOf()>();
51         virtualinvoke $r10.<examples.lollypop.SpeedController: void
52 ajc$after$examples_lollypop_SpeedController$1$fda05aef(examples.lollypop.Factory,int)>(r2, i1);
53
54     label7:
55         return;
56
57         catch java.lang.Throwable from label0 to label1 with label2;
58         catch java.lang.Throwable from label0 to label4 with label5;}}
```

Listing 3: The Jimple partial translation of the RotationMonitor aspect

```
25     public static examples.lollypop.RotationMonitor aspectOf()
26     {
27         examples.lollypop.RotationMonitor $r0, $r3;
28         java.lang.Throwable $r1;
29         org.aspectj.lang.NoAspectBoundException $r2;
30
31         $r0 = <examples.lollypop.RotationMonitor: examples.lollypop.RotationMonitor ajc$perSingletonInstance>;
32         if $r0 != null goto label0;
33
34         $r2 = new org.aspectj.lang.NoAspectBoundException;
35         $r1 = <examples.lollypop.RotationMonitor: java.lang.Throwable ajc$initFailureCause>;
36         specialinvoke $r2.<org.aspectj.lang.NoAspectBoundException: void
37                 <init>(java.lang.String,java.lang.Throwable)>("examples_lollypop_RotationMonitor", $r1);
38         throw $r2;
39
40     label0:
41         $r3 = <examples.lollypop.RotationMonitor: examples.lollypop.RotationMonitor ajc$perSingletonInstance>;
42         return $r3;}}
```

Listing 4: The Jimple translation of the after return advice of the aspect SpeedController

```
1  public class examples.lollypop.SpeedController extends java.lang.Object
2  {
3      public void ajc$afterReturning$examples_lollypop_SpeedController$1$fda05aef(examples.lollypop.Factory, int)
4      {
5          examples.lollypop.SpeedController r0, $r4;
6          examples.lollypop.Factory r1, r2;
7          int i0, i1;
8          examples.lollypop.RotationMonitor $r3;
9
10         r0 := @this: examples.lollypop.SpeedController;
11         r1 := @parameter0: examples.lollypop.Factory;
12         i0 := @parameter1: int;
13         if i0 <= 4 goto label0;
14
15         i1 = i0 / 2;
16         r2 = r1;
17         virtualinvoke r2.<examples.lollypop.Factory: void setRotationSpeed(int)>(i1);
18         $r3 = staticinvoke <examples.lollypop.RotationMonitor: examples.lollypop.RotationMonitor aspectOf()>();
19         virtualinvoke $r3.<examples.lollypop.RotationMonitor:
20           void ajc$afterReturning$examples_lollypop_RotationMonitor$1$839313f3(examples.lollypop.Factory,int)>(r2, i1);
21         $r4 = staticinvoke <examples.lollypop.SpeedController: examples.lollypop.SpeedController aspectOf()>();
22         virtualinvoke $r4.<examples.lollypop.SpeedController:
23           void ajc$afterReturning$examples_lollypop_SpeedController$1$fda05aef(examples.lollypop.Factory,int)>(r2, i1);
24
25      label0:
26         return;
27     }
28
29 }
```

lation and woven automatically together. However, understanding interaction among different aspects is hard and tool support is still very poor.

Our experimental work shows that static analysis of woven code has some potential for making explicit the problems that arise due the complexity of intertwined code. However, simplistic slicing is not sufficient to determine whether two aspects may interfere. In fact, the machinery introduced *for the sake of the weaving itself,* makes slices always overlapping. Thus, our sufficient condition to exclude interference came out to be naive, since it is likely to be always false. Some of the dependencies, caused by the way advice weaving is performed, could be avoided with a parallel source level analysis or a suitable use of dynamic techniques. Smart heuristics are needed, though, and they are likely to depend heavily even on the lowest level of weaver implementation details. A better approach would be the use of annotations by the weaver itself, in order to keep track of the aspect oriented abstraction layer at the bytecode level.

Moreover, slicing Java bytecode also showed us that severe precision problems exist when real world programs are concerned. A common issue is due for example to library methods: consider two calls to the `add` method of two different `Vector`s. Unless the slicer creates multiple copies of the same method to distinguish among different receiving objects, the static analysis will detect spurious dependencies, resulting in large slices. Native code is almost ubiquitous in library frameworks and this means that some dependencies may also be neglected: big slices can even be incomplete! Static analysis of bytecode should be used as a support to further analyses at different levels. Furthermore, the closed world assumption behind any static analysis is challenged by current coding practice. Dynamic linking and reflection are common place in most applications. However the expressive power of intertype declarations common in AspectJ programs forces any analysis to take into account every aspect unit just to compute the static structure of the type system.

The path towards having crosscutting components that can be safely plugged into a system is still long. AspectJ aspects make easy to program quick pools of sparse code and their use spread among developers. However, the next step in dealing with complex cross-cutting concerns and their interaction and evolution needs at least a better tool support.

Acknowledgments

The authors would like to thank Davide Balzarotti and Carlo Ghezzi for their help in clarifying the ideas behind this paper.

5. REFERENCES

[1] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[2] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241*, 1997.

[3] Davide Balzarotti and Mattia Monga. Using program slicing to analyze aspect-oriented composition. In *Proceedings of Foundations of Aspect-Oriented Languages Workshop at AOSD 2004*, 2004.

[4] Davide Balzarotti, Antonio Castaldo D'Ursi, Luca Cavallaro, and Mattia Monga. Slicing AspectJ Woven Code. In *Proceedings of Foundations of Aspect-Oriented Languages Workshop at AOSD 2005*, 2005.

[5] Mark Weiser. Program slicing. *In: IEEE Transactions on Software Engineering, vol. SE-10, no. 4. pp. 352-357*, July 1984.

[6] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 177–184, 1984.

[7] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *SIGPLAN Notices*, 23(7):35–46, June 1988.

[8] Donglin Liang and Mary Jean Harrold. Slicing objects using system dependence graphs. In *ICSM*, pages 358–367, 1998.

[9] Jianjun Zhao. Slicing aspect-oriented software. In *Proceedings of the 10th IEEE International Workshop on Programming Comprehension*, pages 251–260, June 2002.

[10] Venkatesh Prasad Ranganath and John Hatcliff. Slicing concurrent Java programs using Indus and Kaveri, 2006. Tech report. Available at `http://projects.cis.ksu.edu/docman/view.php/12/117/stt05-submission.pdf`.

[11] Santos Academic License. `http://www.cis.ksu.edu/santos/KSUAcademicLicense.shtml`.

[12] Raja Vallée-Rai. Soot: a Java Bytecode Optimization Framework. Master's thesis, McGill University, July 2000.

[13] Ondřej Lhoták. Spark: A flexible points-to analysis framework for Java. Master's thesis, McGill University, December 2002.

[14] Andrew W. Appel. *Modern compiler implementation in Java*. Cambridge University Press, 2002.

[15] Matthew Allen and Susan Horwitz. Slicing Java programs that throw and catch exceptions. In *PEPM '03: Proceedings of the 2003 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 44–54, New York, NY, USA, 2003. ACM Press.

[16] Neil Walkinshaw, Mark Roper, and Murray Wood. The Java System Dependence Graph, September 2003. In Third IEEE International Workshop on Source Code Analysis and Manipulation, page 55.

[17] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 11–20, 1994.

[18] Sheng Liang. *Java(TM) Native Interface: Programmer's Guide and Specification*. Addison-Wesley Professional, 1999.

[19] John Hatcliff, James C. Corbett, Matthew B. Dwyer, Stefan Sokolowski, and Hongjun Zheng. A formal study of slicing for multi-threaded programs with JVM concurrency primitives. In *Static Analysis Symposium*, pages 1–18, 1999.

[20] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In *AOSD'04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 26–35, New York, NY, USA, 2004. ACM Press.

Specializing Continuations A Model for Dynamic Join Points

Christopher J. Dutchyn
Computer Science, University of Saskatchewan
dutchyn@cs.usask.ca

ABSTRACT

By modeling dynamic join points, pointcuts, and advice in a defunctionalized continuation-passing style interpreter, we provide a fundamental account of these AOP mechanisms. Dynamic join points develop in a principled and natural way as activations of continuation frames. Pointcuts arise directly in the semantic specification as predicates identifying continuation frames. Advice models procedures operating on continuations, specializing the behaviour of continuation frames. In this way, an essential form of AOP is seen, neither as meta-programming nor as an ad hoc extension, but as an intrinsic feature of programming languages.

1. INTRODUCTION

Current programming languages offer many ways of organizing code into conceptual blocks, through functions, objects, modules, or some other mechanism. However, programmers often encounter features that do not correspond well to these units of organization. Such features are said to *scatter* and *tangle* with the design of a system, because the code that implements the feature appears across many program units. This scattering and tangling may derive from poor modularization of the implementation; for example, as a result of maintaining pre-existing code. But, recent work[Coady et al., 2004, De Win et al., 2004, Spinczyk and Lohmann, 2004] shows that, in some cases, traditional modularity constructs cannot localize a feature's implementation. In these cases, the implementation contains features which inherently *crosscut* each other.[1] In a procedural language, such a feature might be implemented as parts of disjoint procedures; in an object-oriented language, the feature might span several methods or classes.

These crosscutting features inhibit software development in several ways. For one, it is difficult for the programmer to reason about how the disparate pieces of the feature interact. In addition, they compound development workload because features cannot be tested in isolation. Also, they prevent modular assembly: the programmer cannot simply add or delete these features from a program, since they are not separable units. Aspect-oriented programming (AOP) is intended to provide alternative forms of modularity, to extract these crosscutting features into their own modules. As a result, the code more closely resembles the design. AOP subsumes a number of different modularity technologies, some pre-existing, such as open classes and rewriting systems, and some more unconventional, including dynamic join points and advice. This work provides a novel semantic description of this latter system of dynamic join points, pointcuts, and advice. From this semantics, we provide a new viewpoint to what this form of AOP can modularize well, and eliminate the ad hoc foundation for dynamic join points, pointcuts, and advice.

By modeling dynamic join points, pointcuts, and advice in a defunctionalized continuation-passing style interpreter, we provide a fundamental account of these AOP mechanisms. Dynamic join points no longer rely on intuition to provide "well-defined points in the execution of a program"[Kiczales et al., 2001], but arise in the language semantics in a principled and natural way as activations of continuation frames. Pointcuts arise directly in the semantic specification as predicates identifying continuation frames. Advice models procedures operating on continuations, the dual of its usual behaviour as value transformers. Advice is shown as specializing the behaviour of continuation frames, leading us to understand dynamic join points, pointcuts, and advice as enabling the modularization of control in programs. In this way, an essential form of AOP is seen, neither as meta-programming nor as ad hoc extension, but as an intrinsic feature of programming languages.

We begin our presentation by giving direct semantics for an idealized procedural language, in Section 2. We transform to the continuation passing semantics in Section 3, and identify the three model elements within that semantics in Section 4. Following a comparison of our derivation with other accounts in Section 5, we close with observations on how this work informs our understanding of modularity and provides future avenues of research in Section 6.

[1] Strictly speaking, crosscutting is a three-place relation: we say that two concerns crosscut each other with respect to a mutual representation. The less rigorous 'two concerns crosscut each other' means that they crosscut each other with respect to an implementation that closely parallels typical executable code. Traditional modularity constructs, such as procedures and classes, have a close parallel between source and executable code.

Sixth International Workshop on Foundations of Aspect-Oriented Languages (FOAL 2007) March 13, 2007, Vancouver, BC, Canada.

```
;;program
(define-struct pgm [decls body]) ;PGM (id × decl)* × exp

;; declarations
(define-struct procD [ids body])  ;PROC id* × exp
(define-struct globD [])          ;GLOBAL

;; expressions
(define-struct litX [val])            ;LIT val
(define-struct varX [id])             ;VAR id
(define-struct ifX  [test then else]) ;IF exp exp exp
(define-struct seqX [exps])           ;SEQ exp*
(define-struct letX [ids rands body]) ;LET (id × exp)* exp
(define-struct getX [id])             ;GET id
(define-struct setX [id rand])        ;SET id exp
(define-struct appX [id rands])       ;CALL id exp*

(define-struct pcdX [rands])          ;PROCEED exp*
```

Figure 1: Proc Abstract Syntax

2. A PROCEDURAL LANGUAGE – DIRECT SEMANTICS

As with other semantic presentations (e.g. [Wand et al., 2004]), we choose to work with a first-order, mutually recursive procedural language, PROC. Throughout this paper, our systems are given as definitional interpreters, as introduced by Reynolds [1972], in the style of Friedman et al. [2001]. This interpreter-based approach to modeling various AOP mechanisms originated with our work in the Aspect Sandbox[Dutchyn et al., 2002] and related papers[Masuhara et al., 2003, Wand et al., 2004], and was later adopted by others, including Filman [2001]. For this specific paper, this style of presentation emphasizes the reification of continuations as data structures, thus clarifying our specialization claim.

We begin with the usual syntax and direct-style, big-step semantics, given in Figure 1 and Figure 2 respectively. Programs comprise a set of named mutually-recursive, first-order procedures, and a closed, top-level expression. We assume programs and terms are well-typed. Environments are standard.

One important feature of this definition is that we do not specify the order of evaluation for procedure operands. In particular, we use the Scheme `map` procedure to explicitly provide this non-deterministic behaviour.

We should point out that several usual constructs are present in our syntax, but lacking from our evaluator. This does not impair its expressiveness. In particular, the usual constructs are

- (SEQ $x_1 \ldots$) which evaluates each sub-expression in left-to-right order, yielding the value of the last expression, and
- (LET ([i_1 x_1]$\ldots$) x) which evaluates the body x in an environment enriched with variables i_n bound to the values of the corresponding expressions x_n.

As usual in the literature, these can be denoted in our language the addition of helper procedures as seen in Figure 3. For the sequel, we will employ these notational shorthands.

```
;;; evaluator – expression side
(define (eval x r) ;:(exp × env) → val
  (cond [(litX? x) (litX-val x)]
        [(varX? x) (lookup-env r (varX-id x))]
        [(ifX? x) (eval ((if (eval (ifX-test x) r)
                             ifX-then
                             ifX-else) x) r)]
        [(getX? x) (get-glob (lookup-glob (getX-id x)))]
        [(setX? x) (set-glob (lookup-glob (setX-id x))
                             (eval (setX-rand x) r))]
        [(appX? x) (let ([args (map (lambda (x) (eval x r))
                                    (appX-rands x))]
                         [proc (lookup-proc (appX-id x))])
                     (eval (procV-body v)
                           (extend-env (procV-ids v)
                                       (execF-args f)
                                       empty-env)))]
        [else (error 'eval "not an exp: ~a" x)]))

(define (evlis x* r) ;:(exp* × env) → val*
  (if (null? x*)
      ()
      (cons (eval (car x*) r)
            (evlis (cdr x*) r))))

(define *procs* `([+ . ,(lambda (vs) (+ (car vs) (cadr vs)))]
                  [display . ,(lambda (vs) (display (car vs)) 0)]
                  [newline . ,(lambda (vs) (newline) 0)]))
```

Figure 2: Proc Big-step (Direct) Semantics

(SEQ x_1) ≡ x_1

(SEQ x_1 x_2 ...) ≡ (APP foo i ... x_1)

with helper procedure

(foo . (procV (i ... _) (SEQ x_2 ...)))

where foo is fresh, and each i... are the free variables of the subsequent expressions $x_2 \ldots$

(LET () x) ≡ x

(LET ([i_1 x_1] ... [i_n x_n]) x) ≡ (APP foo i ... x_1 ... x_n)

with helper procedure

(foo . (procV (i ... i_1 ... i_n) x))

where foo is fresh, and each i... are the free variables of the body x

excluding i_1 ... i_n.

Figure 3: Proc Auxiliary Expressions

```
;;; frames
;; auxiliary
(define-struct testF [then else env]) ;TEST exp exp env :: !bool
(define-struct bindF [ids body env])  ;BIND id* exp env :: !val*
(define-struct randF [exp env])       ;RAND exp env :: !val*
(define-struct konsF [vals])          ;KONS val* :: !val
(define-struct rhsF [id])             ;RHS id :: !val
;; effective
(define-struct getF [])               ;GET :: !loc
(define-struct setF [val])            ;SET val :: !loc
(define-struct callF [id])            ;CALL id :: !val
(define-struct execF [args])          ;EXEC val* :: !proc
```

Figure 4: Proc Small-step (cps) Semantics — Continuations

3. A PROCEDURAL LANGUAGE – CONTINUATION SEMANTICS

In order to identify dynamic join points in a principled way, we need to move to a continuation-passing style (CPS) implementation. Continuations, also known as *goto's with arguments*, were first identified by Strachey [2000] and Landin [1998] to model control flow in programs. Later, Reynolds [1993] applied them to ensure that semantics given by definitional interpreters yields a formal model independent of the defining language control constructs.

The CPS transformation[Danvy and Hatcliff, 1993] of our interpreter is systematic, following closely that of Hatcliff and Danvy [1994]. In essence, we treat each of the `let` expressions in the direct `eval` semantics as a *monadic let*[Moggi, 1989, 1991]. These `let`s express a bind operation between the computation of an operand and the computation awaiting that value. Continuations explicitly sequence these bind operations, and reify the computation awaiting the value.

Usually continuations are presented as closures[Danvy, 2000], but Ager et al. [2005] provide an systematic defunctionalization of these closures into tagged structures and an `apply` procedure that gathers the operations of each closure. The only values that pass into the `apply` operation are object-language values (integers, booleans), lists of object-language values (as argument lists), and references (addresses into the store or to procedures). Each tagged structure must contain the values for each variable that the closures reference. The continuation structures required for our small-step interpreter are given in Figure 4.

As usual in operational semantics, we introduce two *auxiliary* continuations, `randF` and `konsF`, to support multiple arguments to procedures. These two continuations provide a strict right-to-left evaluation order for procedure operands. This choice is arbitrary, as explicitly declared in the direct semantics.[2] We could have supplied a non-deterministic ordering in the CPS semantics, introducing other auxiliary continuations; but, that would distract us from our focus. The essential notion is that these supporting continuations have no basis in the direct semantics: they serve only to bridge the gap between the big-step and small-step systems. A third auxiliary continuation, `rhsF` serves the same purpose with regard to the argument to `setX`.

[2]Recall that `map` in Scheme processes the elements in the list in an explicitly undefined order.

Some formalisms avoid this work by silently introducing products or tuple values. Then a polyadic procedure actually accepts a single tuple argument, and explodes the tuple before evaluation of the body. Similarly, procedure applications would contain a hidden tupling action; paralleling our `konsF` continuation behaviour.

Formal, lambda-calculus approaches eliminate the auxiliary continuations by currying procedures and replace polyadic applications with multiple applications. This simplifies the underlying formalism, allowing development of the soundness proofs of the CPS transformation; Thielecke [1997] provides the details.

For our restricted procedural language, the full power of the λ-calculus is not required. Indeed, in the λ-calculus, the `testF` continuation is unnecessary as well. A simple syntactic transformation makes the consequent clauses into thunks (parameterless closures[Danvy and Hatcliff, 1992]). True and False become binary procedures that simply apply one or the other thunk. In summary, we characterize `randF`, `konsF`, and `testF` as *auxiliary* continuations.[3]

The defunctionalized CPS definition of our interpreter is given in Figures 5 and 6.

Our construction is standard, except in three respects. First, we extend Ager's construction to explicitly linearize the continuation. In Ager's construction, each continuation structure, representing a suspended operation awaiting the value of some expression, would contain the rest of the continuation as a field. Only a halt continuation would not have this, as it has nowhere to continue to. In our construction, we represent the entire continuation as a list of frames. A *frame* is a single element in the list representation of the continuation; it indicates the immediate action when this continuation is activated. The remainder of the continuation is in the tail of the list.

- `push` :: $(frm \times cont) \rightarrow cont$ — extends an existing continuation with another frame.
- `pop` :: $(!val \times ((frm \times cont) \rightarrow !val)) \rightarrow cont \rightarrow !val$ — takes a continuation, and either
  - applies the first procedure (`halt`) because the continuation is empty, or
  - applies the second procedure (`step`) to the top continuation frame and the rest of the continuation.

We provide a base definition for `step`, called `base-step` for the language absent aspects. Later we will replace `step` with an aspect-aware version which dispatches appropriately. Also, the halt continuation is represented by the empty list.

The second nonstandard construction is that our implementation *lifts* primitives from the direct interpreter to take the existing continuation as an additional argument. This allows us to provide flow control operations, such as Felleisen's `abort`[Felleisen, 1988], as primitives. This is seen in Figure 7.

Third, our implementation distinguishes the lookup of procedures into a separate continuation, `execF`. Ordinarily, we would require only one continuation, `callF`, to await the evaluation of the operands into argument values. That single continuation would be responsible for locating the de-

[3]These should not be confused with *serious* and *trivial* continuations[Reynolds, 1972], nor with *administrative* continuations[Flanagan et al., 1993].

```
;;; evaluator – expression side
(define (eval x r k) ;: (exp × env × cont) → unit
  (cond [(litX? x) (apply k (litX-val x))]
        [(varX? x) (apply k (lookup-env r (varX-id x)))]
        [(ifX? x) (eval (ifX-test x)
                        r
                        (push (make-testF (ifX-then x)
                                          (ifX-else x)
                                          r)
                              k))]
        [(getX? x) (apply (push (make-getF)
                                k)
                          (lookup-glob (getX-id x)))]
        [(setX? x) (eval (setX-rand x)
                         r
                         (push (make-rhsF (setX-id x))
                               k))]
        [(appX? x) (evlis (appX-rands x)
                          r
                          (push (make-callF (appX-id x))
                                k))]
        [else (error 'eval "not an exp: ~a" x)]))

(define (evlis x* r k) ;: (exp* × env × cont) → unit
  (if (null? x*)
      (apply k
             '())
      (evlis (cdr x*)
             r
             (push (make-randF (car x*) r)
                   k))))

(define (halt v) ;: !val (== val → unit)
  (display v)
  (newline))

(define (apply k v) ;: !(cont × val)
  (((pop halt
         step)
    k)
   v))
```

Figure 5: Proc Small-step (cps) Semantics — Evaluator

```
;;; evaluator – continuation side
(define ((base-step f k) v) ;:(frm × cont) → !val
  (cond ;; auxiliary frames
   [(testF? f) (eval ((if v testF-then testF-else) f)
                     (testF-env f)
                     k)]
   [(randF? f) (eval (randF-exp f)
                     (randF-env f)
                     (push (make-konsF v)
                           k))]
   [(konsF? f) (apply k
                      (cons v (konsF-vals f)))]
   [(rhsF? f) (apply (push (make-setF v)
                           k)
                     (lookup-glob (rhsF-id f)))]
   ;; non-auxiliary frames
   [(getF? f) (apply k
                     (get-glob v))]
   [(setF? f) (apply k
                     (set-glob v (setF-val f)))]
   [(callF? f) (apply (push (make-execF v)
                            k)
                      (lookup-proc (callF-id f)))]
   [(execF? f) (cond [(procV? v)
                      (eval (procV-body v)
                            (extend-env (procV-ids v)
                                        (execF-args f)
                                        empty-env)
                            k)]
                     [(procedure? v) (v (execF-args f) k)]
                     [else
                      (error 'exec "not a procedure: ~a" v)])]
   [else (error 'step "not a frame: ~a" f)]))

(define step base-step)
```

Figure 6: Proc Small-step (cps) Semantics — Evaluator

```
;;; cont ::= frm*
(define (push f k) ;;(frm × cont) → cont
  (cons f k))

(define ((pop e s) k) ;;(!val × ((frm × cont) → !val)) → cont → !val
  (if (null? k)
      e
      (s (car k) (cdr k))))

;;; primitives
(define ((lift p) vs k)
  (apply (p vs) k))

;;; lifted primitives
(define *procs*
  `([+ . ,(lift (lambda (vs) (+ (car vs) (cadr vs))))]
    [cons . , (lift (lambda vs vs))]
    [null? . , (lift (lambda (vs) (null? (car vs))))]
    [display . ,(lift (lambda (vs) (display (car vs)) 0))]
    [newline . ,(lift (lambda (vs) (newline) 0))]
    [abort . ,(lambda (vs k) (apply (car vs) '()))]))

(define (run s)
  (let ([g (parse-prog s)])
    (set! *procs* (cons (PGM-decls g) *procs*))
    (eval (PGM-body g) empty-env '())))
```

Figure 7: Proc Small-step (cps) Semantics — Primitives

sired procedure and initiating the evaluation of it's body-expression with the desired bindings.

Examining the direct semantics closely, we can see that there are two `let` bindings present in the case of an `APP` expression. Other one-step[Danvy and Nielson, 2003] and A-normal[Flanagan et al., 1993] transformations optimize portions of this transformation, usually the second binding. Our more naïve approach allows us to expose the two separate operations, which will be valuable as we extend the system to incorporate dynamic join points, pointcuts, and advice.

4. EXPOSING AOP CONSTRUCTS

With these preliminaries, we are prepared to expose the latent dynamic join points in Proc, and provide syntax to denote pointcuts and advice. We need to describe three items (quoted from [Kiczales et al., 1997]):

1. **dynamic join points** — "principled points in the execution". These will be states in the interpreter where values are applied to non-auxiliary continuation frames.

2. **pointcuts** — "a means of identifying join points". These will be syntax for predicates over the value and continuation frame content.

3. **advice** — "a means of affecting the semantics at those join points". This is implemented as the advice body as a procedure applied to the continuation frame.

We will examine each of these in turn.

4.1 Dynamic Join Points

Dynamic join points are the first abstraction in our model. Other semantic models simply list dynamic join points without supporting the intuition for their selection. The underlying principles are not enunciated. Identifying this principle is a key result of this work.

For us, join points are activations of certain continuation frames. Recall that we introduced auxiliary frames to support our eager, right-to-left evaluation order in the cps semantics. Therefore, we adopt the following principle:

> A dynamic join point is modeled as a state in the interpreter where a non-auxiliary continuation frame is applied to a value.

Auxiliary continuation frames do not correspond to principled points in the execution of a program. For example, our `konsF` and `randF` frames were arbitrarily chosen to supply an eager, right-to-left evaluation order. With a lazy big-step semantics, or with a different evaluation order, different auxiliary continuation frames would be required. Similarly, the `testF` frame exists to postpone the choice of alternatives to an `ifX` until the test has been evaluated first. The `rhsF` and bindF auxiliary frames exist to support the single reduction ordering that cps interpreters must support – again they are not mandated by the big-step semantics.

Therefore, in Proc, we have four frames corresponding to dynamic join points:

- `callF` ($id \vdash !val^*)$) — takes an procedure name and constructs a frame that will consume a list of argument values and apply the named procedure to them,
- `execF` ($val^* \vdash !proc$) — stores a list of argument values and constructs a frame that will consume a procedure and apply it to the list of values,
- `getF` ($id \vdash !loc$) — takes an identifier and constructs a frame that will consume a store location and continue with its content,
- `setF` ($val \vdash !loc$) — takes a value and constructs a frame that will consume a store location and continue after updating its content.

The type signatures indicate the type of the information stored in the continuation frame, followed by type of the continuation once the frame is `push`ed. We use negative types for continuations, in keeping with previous workJouvelot and Gifford [1989], Murthy [1992]. Thielecke [1997] explores this in detail.

In each case, a dynamic join point has various items of information available, some from the value applied to the continuation, some from the frame itself. These include

1. a procedure, either by name (in the case of `callF`) or as an actual structure (in the case of `execF`),
2. a list of values corresponding to the arguments to the procedure (in the case of `callF` or `execF`,
3. a value and a store reference (in the case of `setF`),
4. a store reference (in the case of `getF`).

Our join points are summarized in Table 1.

In our model, dynamic join points make accessible the latent control structure of the language semantics. Dynamic

Dynamic Join Point		
Value Consumed		Frame Information
(loc i_{global})	▶	(getF)
(loc i_{global})	▶	(setF val)
(val*)	▶	(callF i_{proc})
(proc i_{proc})	▶	(execF val*)

Table 1: Dynamic Join Points

```
;;; pointcuts

;; effective continuation frame matching
(define-struct getC [gid])       ; GETPC id
(define-struct setC [gid id])    ; SETPC id id
(define-struct callC [pid ids]) ; CALLPC id id*
(define-struct execC [pid ids]) ; EXECPC id id*

;; combinational
(define-struct orC [pcs])        ; ORPC pcut*
(define-struct notC [pc])        ; NOTPC pcut
```

Figure 8: Proc Pointcuts — Abstract Syntax

join points correspond to continuation frames, and are modeled by states within the interpreter. Our set of dynamic join points is stable with regard to semantic changes such as altering the order of evaluation, or moving from eager to lazy evaluation.[4] Other semantic changes involved in extending the big-step semantics, notably introducing new terms (e.g. for-loops[Harbulot and Gurd, 2006]), would introduce or modify the set of dynamic join points.

Our dynamic join points systematically align with points in the model that are well-accepted as being semantically meaningful. Our principle defines this systematic alignment. In other models, some have framed dynamic join points as program rewrite points[Aßmann and Ludwig, 1999, Roychoudhury and Gray, 2005]. Other accounts have dynamic join points appear as an ad hoc list, including in our earlier work[Wand et al., 2004]. Our principled approach provides a more robust and elegant description.

4.2 Pointcuts

The second abstraction we must add to our model is pointcuts. Pointcuts are syntax that provide a means to identify our dynamic join points. We have a pointcut for identifying each kind of continuation frame (join point): `call`, `exec`, `get`, and `set`. We adopt the following syntax for pointcuts. It contains four structures, one for each kind of dynamic join point.

We have chosen a direct pointcut syntax, where the procedure name and the argument names are given directly in the pointcut. In the next section, we will use the argument names to offer access to the arguments in the advice. The semantics of a pointcut is to examine whether the current interpreter state matches the identified continuation frame – both in kind and content – and the current value. This is seen in Figure 9.

In the case of a `callC` pointcut, we ensure that the frame is a `callF` frame, and that it holds a procedure name equal

[4]Changing to lazy evaluation would alter the order that join points are encountered during the evaluation.

```
;;; matching
:MATCH id* val* (val* → (val × frm))
(define-struct match [ids vals prcd])

(define (match-pc c v f) ;:(pcut × val × frm) → match
  (cond ;; combinational pointcuts
        [(orC? c) (let loop ([pcs (orC-pcs c)])
                    (if (null? pcs)
                        #f
                        (or (match-pc (car pcs) v f)
                            (loop (cdr pcs)))))]
        [(notC? c) (if (match-pc (notC-pc c) v f)
                       #f
                       (make-match '()
                                   '()
                                   (lambda (nv)
                                     (values v f))))]
        ;; fundamental pointcuts
        [(getC? c) (and (getF? f)
                        (eq? (lookup-glob (getC-gid c)) v)
                        (make-match '()
                                    '()
                                    (lambda (nv)
                                      (values v f))))]
        [(setC? c) (and (setF? f)
                        (eq? (lookup-glob (setC-gid c)) v)
                        (make-match `(,(setC-id c))
                                    `(,(setF-val f))
                                    (lambda (nv)
                                      (values v
                                              (make-setF
                                               (car nv))))))]
        [(callC? c) (and (callF? f)
                         (eq? (callC-pid c) (callF-id f))
                         (make-match (callC-ids c)
                                     v
                                     (lambda (nv)
                                       (values nv f))))]
        [(execC? c) (and (execF? f)
                         (eq? (lookup-proc (execC-pid c)) v)
                         (make-match (execC-ids c)
                                     (execF-args f)
                                     (lambda (nv)
                                       (values v
                                               (make-execF
                                                nv)))))]
        [(advC? c)]
        [else (error 'match-pc "not a pointcut: ~a" c)]))
```

Figure 9: Proc Pointcuts — Implementation

to the one given in the pointcut. For a `execC` pointcut, we ensure that the frame is an `execF` frame and that the supplied value is a procedure whose name is equal to the one given in the pointcut. `GetC`, and `setC` pointcuts are similar, matching the `getF` and `setF` frames respectively.

We also include two combinational pointcuts. The first is `orC`, which matches any dynamic join point which matches the first subpointcut; or, failing that, matches the second subpointcut. This allows us to abstract a concern that cuts across multiple procedures. For example, one might consider two `displayX` procedures, each with a different output format, to be a single display concern.

This combinational pointcut provides a simple specialization ordering to pointcuts; and, by extension, advice. Any given pointcut, `A`, is more specialized than `orC(A B)` for any distinct `B` pointcut. Pointcuts do not have a unique total ordering, only a partial order. They can be totally ordered using the standard topological sort. By extension, advice can be ordered by this total pointcut order.

The other combinational pointcut is `notC` which simply matches every join point which differs from its subpointcut. It returns no matched values, it simple succeeds or fails.

A pointcut matches the top continuation frame, the list of identifiers from the pointcut is returned. If a match is not found, `#f` (Scheme `false`) is returned. In our implementation, matching against a `orC` pointcut yields the identifiers for the matching sub-pointcut. This means that each subpointcut must provide the same identifiers.

In our model, we adopt the principle that

> pointcuts do not alter the semantic behaviour of the program or language.

In our system, advice is solely responsible for altering behaviour at join points. This leads to concerns with contextual pointcuts.

4.2.1 Contextual Pointcuts

It is tantalizing to consider the entire continuation for the purposes of matching join points. If we did this, then we can quickly and easily provide the various contextual pointcuts, including a novel one:

- (`cflowbelow pcut`): climb down (towards older) the list of frames, skipping the current frame,
- (`cflow pcut`): equivalent to (`or pcut (cflowbelow pcut)`).
- (`cflowabove pcut`): climb back up (towards newer) the list of frames, skipping the current frame.[5,6]

These contextual pointcuts provide a mechanism for characterizing join points based on their temporal context in the control flow. The usual `cflow` and `cflowbelow` provide the usual "within another control context" recognizer by searching downward toward the program start. Our novel `cflowabove` construct provides a way to search in the other direction, "encloses another control context". This is useful, along with the `not` pointcut, to provide the equivalent to Prolog cuts in the context search.

Unfortunately, in a language with tail call optimization, this simplistic implementation does not work. The context of interest may be removed from the continuation frame list by the tail call optimization, and the desired advice will not be triggered. In fact, deeper consideration of the contextual pointcuts convinces us that these pointcuts actually have a computational effect: they require the evaluator to remember where the exit from the interesting context occurs. This is conveniently simple in non-tail call languages: popping the identified continuation frame can serve as the marker.

If we know the pointcuts in advance, we can avoid having the pointcut alter all matching frame behaviour by retaining only the interesting frames, the ones identified in `cflow` pointcuts, on the stack. This requires advance knowledge; but can then be implemented quite efficiently[Clements and Felleisen, 2004].

But, tail call languages require some additional mechanism — context may disappear before related advice is triggered. One solution might be to include some special context-marking continuation frame – these are called *continuation marks*[Clements and Felleisen, 2004], and essentially provide a safe-for-space implementation of dynamic binding. This is the mechanism applied in the AspectScheme language[Dutchyn et al., 2006].

We choose not to add new mechanisms, and wish to cleave to the principle that pointcuts do not change the language semantics nor the program behaviour. Therefore, we must supply separate implementations of these pointcuts. Fortunately, Masuhara et al. [2003] provides a state-based `cflow` design. It can be modelled in our language as two coordinated pieces of advice. The first specialises join points matching the control flow of interest to push the arguments onto a stack data structure, `proceed` to determine the result, pop the stack, and return that result. The second specialises the join points of interest within the control flow to check for available context on the stack data structure, and modify the continuation behaviour appropriately.

In our model, pointcuts are first-order predicates for dynamic join points. In this general view, we are no different from other accounts of dynamic join points, pointcuts, and advice AOP. But, pointcuts identify continuation frames at which advice bodies are to operate. Hence, we can view advice as extending and specializing the behaviour of control points in programs.

4.3 Advice

Now we come to the third feature of our model — advice. A piece of advice needs to specify a means of affecting the semantics at join points. Syntactically, it contains two parts:

1. a pointcut — which indicates which dynamic join points are to be affected
2. an advice body — an expression

The new syntax element for advice declarations is given in Figure 10. Advice are declarations in our model, just like procedures. Therefore, they will be have identifiers bound to them, just like procedures do.

[5]With cactus stacks[Clinger et al., 1999] for threaded languages, this requires the correct path back up to be maintained.

[6]Pointcuts containing `cflowabove` can be rewritten using `cflowbelow` alone. This is clear by recognizing that pointcuts form a regular language describing stack structures[Sereni and de Moor, 2003], and that `cflowbelow` and `cflowabove` are the left- and right-regular descriptions. Of course, `cflowabove` is expressive in the sense of Felleisen [1991] because the tranformation requires a global rewrite of the entire pointcut.

```
;;; declarations
(define-struct advD [pc body]) ;DECL +:= ADVICE pcut exp
```

Figure 10: Proc Advice Declaration – Abstract Syntax

(BEFORE pcx) ≡ (AROUND pc (SEQ x proceed))

(AFTER pc x) ≡ (AROUND pc (APP foo (proceed)))

with fresh helper procedure

(foo . (procV (v) (SEQ x v)))

Figure 11: Proc Before and After Advice

In our system, all advice is `around` advice. That is, it has control over, and alters the behaviour of, the underlying dynamic join point. Our advice may `proceed` that dynamic join point zero, one, or many times. This does not restrict the generality of our model, as common `before` and `after` advice are the two possible orderings of the advice body and `proceed`, shown in Figure 11.

Semantically, an advice resembles a procedure. The pointcut part identifies the affected dynamic join points, and provides binding names for the arguments of the dynamic join point. In our model the advice body acts like a procedure body, but its locus of application differs.

A procedure is usually applied to some values to yield another value. For example, the procedure `pick` in the following code:

```
(define (pick b) (if x 1 2))

(+ (pick #t) 3)
```

is applied to `#t` to yield a new value `1`. Filinski [1989] first recognized that `pick` transforms the continuation of the procedure application from

```
(lambda (n)   ; await number, add three, halt
  (+ n 3))
```

to

```
(lambda (b)   ; await boolean
  (let ([n (if b 1 2)])   ; select number
    ((lambda (n) (+ n 3)) ; original continuation
     n)))     ; given the selected number
```

One way to discern this different mode of application is to consider the types of the elements involved. Jouvelot and Gifford [1989] recognized that the type of the original continuation is !number (read as consumes number), and that applying `pick` has extended the continuation to consume a boolean (typed !boolean). `Pick` has type boolean → number when considered as a value transformer, and has type !number →!boolean as a continuation transformer[Strachey, 2000].

In Filinski's *symmetric lambda calculus*[Filinski, 1989], procedures could be applied in either way: to values, yielding new values; or to continuations, yielding new continuations. In our model, advice provides this similar procedure application to continuations. We present our semantics in five parts – advice elaboration and matching, altered `step/prim` to support advice execution, a new `step/weave` to weave advice into the execution of the program, advice invocation, and last, the `proceed` expression.

First, we recognize that advice is a declaration; hence we need to elaborate the advice declarations, in the same trivial way we did for procedure declarations. This is displayed in Appendix A.

Matching is also shown in Figure 12. We simply walk the elaborated list of advice, comparing the pointcuts and returning a `match` containing the pointcut-match identifiers and the advice itself. It also contains details on how to `proceed`, but we will examine those later.

Pointcuts not only provide parameters at the application site, but also automate the application of advice to all matching dynamic join points. This universal application of advice extends the semantics of matching dynamic join points to contain additional behaviour.

A subtle difference is that advice can extend the behaviour of a join point, by calling `proceed`, a new expression in our Proc language. It takes a set of arguments and passes them on to the next advice, or the underlying dynamic join point if all advice has been invoked. The syntax for `proceed`, as well as the extension of `eval` is given in Figure 13.

In order for `proceed` to work, we need to provide the remaining matched advice, and a representation of the original join point. This is done by binding a special variable, '`%proceed` into the environment for the advice. It contains the remaining advice, if any, and the original procedure name (in the case of a `callF` dynamic join point), the original `procV` or procedure (in the case of an `execF` dynamic join point).

Recalling our principle that dynamic join points correspond to frame activations, we recognize that our new frame, `advF` defines a new set of dynamic join points that may be matched against. By construction, all of our declarations are bound to identifiers, advice declarations will also have names. Hence, we naturally provide an advice execution dynamic join point, and its associated matching operation. By construction, all activations of `advF` frames are processed by `adv-step`, so the weaving of additional behaviour is automatic. The call structure that makes this so is:

- `apply` calls `adv-step`
- `adv-step` looks for matching advice
  - if there is none, `base-step` provides the fundamental behaviour of the dynamic join point
  - if there are matches, we evaluate arguments and push an advice execution dynamic join point

```
;;advice matching against frames/join points
(define (((adv-step advs) f k) v) ;:adv* → (frm × cont) → !val
  (let loop ([advs advs])
    (cond [(null? advs) ((base-step f k) v)]
          [(match-pc (caar advs) v f) =>
           (lambda (m)
             (eval (cdar advs)
                   (extend-env `(%proceed %advs . ,(match-ids m))
                               `(,(match-prcd m)
                                 ,(cdr advs) . ,(match-vals m))
                               empty-env)
                   k))]
          [else (loop (cdr advs))])))

(define step adv-step) ;;redefinition
```

Figure 12: Proc Advice – Matching

```
;;; proceed needs a new advice-execution frame
;; – hence, a new join point
ADV (val* rightarrow val+frm) adv* :: !val
(define-struct advF [v->v+f advs])

;;; evaluator – expression side
(define (eval x r k) ;:(exp × env × cont) → unit
  (cond ...
        [(pcdX? x) (evlis (pcdX-rands x)
                          r
                          (push
                           (make-advF (lookup-env r '%proceed)
                                      (lookup-env r '%advs))
                           k))]
        [else (error 'eval "not an exp: ~a" x)]))

;;; evaluator – continuation side
(define ((base-step f k) v) ;:(frm × cont) → !val
  (cond ...
        ;; non-auxiliary frames
        ...
        [(advF? f)
         (let-values ([(v1 f1) ((advF-v->v+f f) v)])
                     (((adv-step (advF-advs f)) f1 k) v1))]
        [else (error 'step "not a frame: ~a" f)]))
```

Figure 13: Proc Advice – Proceed

- `proceed` expressions do the same to extract the next advice or the final dynamic join point and initiate it's execution.

In our model, an advice body provides new behaviour for each dynamic join point (control point) identified by the advice' pointcut. This new behaviour *extends* the original because advice may contain additional program operations. This new behaviour *specializes* the original because the original behaviour is available through the `proceed` expression.

5. COMPARISON TO OTHER SEMANTICS

We compare our dynamic join point schema to those of other semantic models. The first two are semantic models are joint work between this author and others.

5.1 Aspect Sandbox

In joint work, Dutchyn et al. [2002] and Wand et al. [2004], this author developed a number of semantic models of aspect-oriented programs, both for object-oriented and procedural languages. That work provides a model of a first-order, mutually-recursive procedural programming language. In that semantic model, three kinds of dynamic join points were constructed ex nihilo: `pcall`, `pexecution`, and `aexecution`. This work develops the principle behind the intuition of those three dynamic join point kinds.

Our model also eliminates some of the irregularities in these other implementations. For instance, because Wand et al. [2004] implements a direct semantics, it maintains a separate stack of dynamic join points rather than relying on structured continuations to do this. Further, it relies on thunks to delay execution of `proceed`; in our semantics, this arises within from the continuation structure.

We focus on the core semantic model for our system, therefore we have avoided the more extensive pointcut languages found in mainstream languages. We adopt conventions from early versions of AspectJ[Kiczales et al., 2001]. Current version of AspectJ provides a pointcut calculus with separate binding combinators (e.g. `args`, and `target`), as well as pattern matching and other features. In our model, `&&` provides no additional expressive power, so we do not include it.

Some Aspect Sandbox pointcuts are lexical, such as `within` which restricts join point matches to those which occur during the evaluation of the expressions within a specific procedure. It can be characterized as join points which appear with no intervening frames; as this is the situation where lexical and dynamic scoping coincide. But, this pointcut is strongly dependent on the textual representation of the program. As a result, programmers can easily re-modularize or abstract their code to retain what appears to be the same join point sequencing; but unwittingly introduce or eliminate join points from those identified by this pointcut.

`Within` is dramatically at odds with the dynamic join point and advice model. Indeed, it can mislead programmers into believing that dynamic join points are expressions and that aspect-oriented programming is simply a program generation/rewriting technique. Our model shows how join points and advice aspects arise from the *semantics* of a language, not from the *syntax* of a language. `Within` is possible wiht our framework, based on the observation that lexical scope coincides with dynamic scope, until another lexical scope intervenes. So, `within` can be implemented similar to our `cflow` example, with a third piece of advice that masks the stack data structure once another lexical scope is entered. The join points identifying a new scope are the execution join points; available through the `exec` pointcut.

In summary, our pointcut language provides is a reasonable fit for our model approach.

5.2 AspectScheme

The author contributed the semantic description of AspectScheme[Dutchyn et al., 2006] and the online implementation[Dutchyn, 2006]. AspectScheme models join points as procedure applications in context of other in-progress procedure applications. It depends on novel *continuation marks* to express the structure of the continuation stack, and relies on macros to provide weaving whenever a procedure is applied. This is practical solution for extending Scheme, where continuations are available only as opaque procedures—their structure cannot be examined. This work simplifies the AspectScheme semantic presentation to recognize that continuation marks are not required, provided Ager et al. [2005]'s defunctionalized continuation model is available.

AspectScheme offers only a single kind of dynamic join point, a procedure application in the context of pending procedures. This corresponds to our `execF` dynamic join point, but with additional context. But, because dynamic join points are first-class objects, temporally ordered lists of procedures and arguments in AspectScheme, the programmer can extend the set of pointcuts by writing their own. This expressiveness is put to good use in showing practical applications of advice.

5.3 PolyAML and μABC

Dantas et al. [2005] provide a PolyAML, a polymorphic aspect-oriented programming language. It is implemented in two levels, a polymorphic surface syntax, which is translated into a monomorphic dynamic semantics, $\mathbb{F}_A$. Their focus is on type-checking, and `around` aspects are incompatible with that goal. They can only support oblivious[Filman and Friedman, 2004] aspects, which must be `before` and `after`

only. A later paper,[Dantas et al., to appear] solves the typing difficulties with `around` advice, using novel local type inference techniques.

Their monomorphic machine is described in terms of context semantics[Ager et al., 2003]. Briefly, a context is an expression with a hole, which the current redex will plug, once it reduces to a value. The machine shifts into deeper and deeper contexts until values can be directly computed, either as literals or variable references. Once all the holes are plugged in a redex, it is reduced to a value and plugged into its pending context. Danvy et al. investigated the equivalence between context semantics and continuation semantics. PolyAML's label method for providing aspects in monomorphic context semantics appears to be equivalent to AspectScheme's continuation marks in a continuation semantics.

It would be interesting to attempt to remove the labels from their $\mathbb{F}_A$ core calculus by reifying the actual continuation structures. We expect that the principled set of dynamic join points would again become apparent, rather than imposed externally.

Bruns et al. [2004] provides an untyped core calculus for aspects. As Dantas et al. [2005] note, this core calculus strongly resembles their $\mathbb{F}_A$ monomorphic context semantics. Again, labels are used to annotate a context and provide an understanding of dynamic join points. They support full `around` advice, but make no attempt to supply static type checking or inference.

5.4 Other Related Work

Several other semantic formulations for aspects have been offered.

Douence et al. [2001] considers dynamic join points as events, and provides oblivious aspects. This is done by providing a custom sequencing monad that recognizes computations, and wraps them with the additional behaviour of the advice. Unfortunately, this is insufficient to allow `around` advice to alter the parameters of the wrapped computation. Only the option to `proceed` with the original arguments is available.

Andrews [2001] provides a process-calculus description of aspects. Oblivious aspects are provided. But, constrained by encapsulated processes, full `around` aspects are not possible.

Clifton and Leavens [2006] further explored the idea of split `call` and `exec` join points; a distinction that originated in Wand et al. [2002].

Kojarski and Lorentz [2005] consider composition of multiple aspect extensions to a base language. Our work indicates that, for pointcut-and-advice aspects, there seems to be a natural extension, which modularizes over the dynamic semantic of the language. Preliminary work suggests that the same principled approach can be applied to other phases [Cardelli, 1988] in a programming language, leading to several other extensions such as static join points (as in AspectJ). The compositional behaviour of these different phases remains to be explored.

Endoh et al. [2006] also use the CPS transformation to expose join points; their interest is to reduce the number of advice kinds (e.g. `after returning`). This work aims at a more fundamental understanding of pointcuts and advice AOP, and attempts to expose the principles underlying it. As a result, our work identifies an additional join point, the advice-execution join point (which AspectJ provides without explanation). Furthermore, we takes view that advice specialize continuation behaviour, leading to a principled understanding of the modularity that AOP can provide.

6. SUMMARY

Aspect-oriented programming (AOP) is crosscutting-modularity technology. It comes in a variety of forms, including open classes and dynamic join points. The former example provide separation of concerns that involve data modularity. This paper demonstrates that the latter provides separation of concerns invested in modularizing (identifying, specializing and isolating) control structures. Although unsurprising, this characterization of different kinds of aspects based on what they modularize is powerful. It fundamentally sharpens our understanding *what dynamic aspects are*, and therefore enables us to construct and apply them effectively.

Our construction has intriguing parallels with object-oriented programming. From one perspective, objects provide a way for programmers to group related data fields, tagging them so that late-bound operations can be supplied. Our construction appears to generalize to grouping related continuation frames, tagging them so that late-bound operations can be supplied.

One example of this sort of aspect hierarchy would be a base aspect that provides the state-based implementation of `cflow`. By extending that abstract aspect with two pointcuts and the desired advice, we provide modular instances of `cflow` and its ilk; thus eliminating them from the base language. The language remains expressive and becomes simpler.

The duality between values and continuations may offer some insight into what dynamic join point-based AOP is effective at modularizing. We believe that a full-fledged aspect, intended to capture a crosscutting feature, combines multiple pointcuts and advice into an *abstract control type* paralleling ubiquitous abstract data type. Our implementation highlights the existence of a dispatch to late-bound advice specializing the behaviour of that continuation frame. Can this be extended to give a unified understanding of aspects and objects as dual similar to value and continuation duality?

Furthermore, our construction suggests that an appropriate type theory for dynamic join points should be built on the ones for continuations. In particular, we are investigating the negative types of [Griffon, 1990, Murthy, 1992] which characterize continuations, and the more recent work of Shan [2003] and Biernacki et al. [2006] who look at polarized types for delimited continuations[Biernacka et al., 2005, Shan, 1999] of which our frames are a degenerate example.

In summary, our work provides a well-founded implementation of aspects with three key properties:

1. Dynamic join points, pointcuts, and advice aspects are modeled directly in continuation semantics; without the need for extraneous labels or continuation marks,

2. Principled dynamic join points arise naturally, as continuation frames, from describing programming languages in continuation semantics, and

3. Advice acts as a procedure on these continuation frames, providing specialized behaviour for them.

We give a formal model of dynamic join points, pointcuts, and advice built on the well-understood processes of conversion to continuation-passing-style, and defunctionalization. We demonstrate that dynamic join points arise naturally in this formulation, as continuation frames. Therefore, advice can specialize their behaviour directly in our construction. Furthermore, we demonstrate that, in our model, `cflow` corresponds to a continuation context, and interacts poorly with tail-call optimizations, but can be recognized as a state effect.

In this way, we provide a fundamental account of these AOP mechanisms that arises naturally from the semantic description of the language. Our model is by construction, not ad hoc. Our model does not entail pre-processing or other meta-programming techniques.

References

M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. A functional correspondence between evaluators and abstract machines. In *International Conference on Principles and Practice of Declarative Programming*, pages 8–19. ACM Press, Aug. 2003. ISBN 1-58113-705-2.

M. S. Ager, O. Danvy, and J. Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theoretical Computer Science*, 342(1):149–172, 2005.

J. H. Andrews. Process-algebraic foundations of aspect-oriented programming. In A. Yonezawa and S. Matsuoka, editors, *Workshop on Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192, pages 187–209. Springer-Verlag, Sept. 2001. ISBN 3-540-42618-3.

A. Aßmann and A. Ludwig. Aspect weaving by graph rewriting. In U. Eisenecker and K. Czarnecki, editors, *International Symposium on Generative Component-based Software Engineering*, Oct. 1999.

M. Biernacka, D. Biernacki, and O. Danvy. An operational foundation for delimited continuations in the cps hierarchy. Technical Report RS–05–25, BRICS, University of Aarhus, Aug. 2005. URL `http://www.brics.dk/RS/05/24/BRICS-RS-05-24.pdf`. to appear in Logical Methods in Computer Science.

D. Biernacki, O. Danvy, and C. chieh Shan. On the static and dynamic extents of delimited continuations. *Science of Computer Programming*, 60(3):274–297, 2006.

G. Bruns, R. Jagadeesan, A. Jeffrey, and J. Riely. μABC: A minimal aspect calculus. In P. Gardner and N. Yoshida, editors, *International Conference on Concurrency Theory*, volume 3170 of *Lecture Notes in Computer Science*, pages 209–224. Springer-Verlag, Sept. 2004. ISBN 3-540-22940-X.

L. Cardelli. Phase distinctions in type theory. Manuscript, 1988. URL `citeseer.ist.psu.edu/cardelli88phase.html`.

J. Clements and M. Felleisen. A tail-recursive machine with stack inspection. *ACM Transactions on Programming Languages and Systems*, 26(6):1029–1052, 2004.

C. Clifton and G. T. Leavens. MiniMAO: An imperative core language for studying aspect-oriented reasoning. *Sci. Comput. Programming*, 63(3):321–374, Dec. 2006.

W. D. Clinger, A. H. Hartheimer, and E. M. Ost. Implementation strategies for first-class continuations. *Higher-Order and Symbolic Computation*, 12(1):7–45, Apr. 1999.

Y. Coady, G. Kiczales, M. Feeley, N. Hutchinson, and J. S. Ong. *Structuring Operating System Aspects*, chapter 28, pages 651–657. In , Filman et al. [2004], Oct. 2004.

D. S. Dantas, D. Walker, G. Washburn, and S. Weirich. PolyAML: A polymorphic aspect-oriented functional programming language. In O. Danvy and B. C. Pierce, editors, *International Conference on Functional Programming*, Sept. 2005. ISBN 1-59593-064-7.

D. S. Dantas, D. Walker, G. Washburn, and S. Weirich. AspectML: A polymorphic aspect-oriented functional programming language. *Transactions on Programming Languages and Systems*, to appear.

O. Danvy. Formalizing implementation strategies for first-class continuations. In G. Smolka, editor, *European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 88–103. Springer-Verlag, Mar. 2000. ISBN 3-540-67262-1.

O. Danvy and J. Hatcliff. Thunks (continued). In *Workshop on Static Analysis*, pages 3–11, 1992.

O. Danvy and J. Hatcliff. On the transformation between direct and continuation semantics. In S. D. Brookes, M. G. Main, A. Melton, M. W. Mislove, and D. A. Schmidt, editors, *Conference on Mathematical Foundations of Programming Semantics*, volume 802 of *Lecture Notes in Computer Science*, pages 627–648. Springer-Verlag, Apr. 1993. ISBN 3-540-58027-1.

O. Danvy and L. R. Nielson. A first-order one-pass cps transformation. *Theoretical Computer Science*, 308(1–3): 239–257, Nov. 2003.

B. De Win, W. Joosen, and F. Piessens. *Developing Secure Applications Through Aspect-Oriented Programming*, chapter 27, pages 633–560. In , Filman et al. [2004], Oct. 2004.

R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *Lecture Notes in Computer Science*, volume 2192, pages 170–186, Sept. 2001.

C. J. Dutchyn. AspectScheme v. 2. PLaneT repository, Jan. 2006. URL `http://planet.plt-scheme.org/300/#aspect-scheme.plt2.1`.

C. J. Dutchyn, G. Kiczales, and H. Masuhara. Aspect Sandbox. internet, 2002. URL `http://www/labs/spl/projects/asb.html`.

C. J. Dutchyn, D. B. Tucker, and S. Krishnamurthi. Semantics and scoping of aspects in higher-order languages. *Science of Computer Programming*, 63(3):207–239, Dec. 2006.

Y. Endoh, H. Masuhara, and A. Yonezawa. Continuation join points. In C. Clifton, R. Lämmel, and G. T. Leavens, editors, *Workshop on Foundations of Aspect Oriented Languages*, pages 1–10, Mar. 2006. Iowa State University TR#06-01.

M. Felleisen. The theory and practice of first-class prompts. In *Symposium on Principles of Programming Languages*, pages 180–190, 1988.

M. Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17:35–75, 1991.

A. Filinski. Declarative continuations and categorical duality. Master's thesis, DIKU, University of Copenhagen, Aug. 1989.

R. Filman. Understanding AOP through the study of interpreters, 2001. URL `citeseer.ist.psu.edu/571298.html`.

R. Filman and D. Friedman. *Aspect-Oriented Programming is Quantification and Obliviousness*, chapter 2, pages 21–36. In , Filman et al. [2004], Oct. 2004.

R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, Oct. 2004.

C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Conference Programming Language Design and Implementation*, pages 237–247, 1993.

D. Friedman, M. Wand, and C. Haynes. *Essentials of Programming Languages*. MIT Press, 2001.

T. G. Griffon. A formulæ-as-types notion of control. In *Symposium on Principles of Programming Languages*, pages 47–57. ACM Press, January 1990.

B. Harbulot and J. R. Gurd. A join point for loops in AspectJ. In R. Filman, H. Masuhara, and A. Rashid, editors, *Conference on Aspect Oriented Software Development*, pages 63–74. ACM Press, Mar. 2006. ISBN 1-59593-300-x.

J. Hatcliff and O. Danvy. A generic account of continuation-passing styles. In *Symposium on Principles of Programming Languages*, pages 458–471, 1994.

P. Jouvelot and D. K. Gifford. Reasoning about continuations with control effects. In R. L. Wexelblat, editor, *Conference Programming Language Design and Implementation*, pages 218–226. ACM Press, June 1989. ISBN 0-89791-306-X.

G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, June 1997.

G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *European Conference on Object-Oriented Programming*, 2001.

S. Kojarski and D. H. Lorentz. Pluggable AOP – designing aspect mechanisms for third-party composition. In R. P. Gabriel, editor, *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 247–263. ACM Press, Oct. 2005. ISBN 1-59593-031-0.

P. J. Landin. A generalization of jumps and labels. *Higher-Order and Symbolic Computation*, 11(2):125–143, 1998.

H. Masuhara, G. Kiczales, and C. J. Dutchyn. A compilation and optimization model for aspect-oriented programs. In G. Hedin, editor, *International Conference on Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 46–60. Springer-Verlag, January 2003.

E. Moggi. Computational lambda-calculus and monads. In *Logic in Computer Science*, pages 14–23. IEEE, June 1989.

E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

C. R. Murthy. A computational analysis of girard's translation and LC. In *Symposium on Logic in Computer Science*, pages 90–101. IEEE, June 1992.

J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM National Conference*, pages 717–740. ACM Press, 1972.

J. C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3-4):233–248, 1993.

S. Roychoudhury and J. Gray. AOP for everyone – cracking the multiple weavers problem. Manuscript, 2005. URL `http://www.cis.uab.edu/gray/Pubs/software-suman.pdf`.

D. Sereni and O. de Moor. Static analysis of aspects. In *ACM SIGPLAN Conference on Aspect-oriented Software Development*, pages 30–39, 2003.

C.-c. Shan. From shift and reset to polarized logic. Manuscript, 2003. URL `http://www.eecs.harvard.edu/čcshan/polar/paper.pdf`.

C.-c. Shan. Shift to control. In O. Shivers and O. Waddell, editors, *Scheme Workshop*, 1999.

O. Spinczyk and D. Lohmann. Using AOP to develop architecture-neutral operating system components. In *SIGOPS European Workshop*, pages 188–192. ACM Press, Sept. 2004.

C. Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13 (1/2):11–49, 2000.

H. Thielecke. *Categorical Structure of Continuation Passing Style*. PhD thesis, University of Edinburgh, 1997. Also available as technical report ECS-LFCS-97-376.

M. Wand, G. Kiczales, and C. J. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. In G. T. Leavens and R. Cytron, editors, *Workshop on Foundations of Aspect Oriented Languages*, pages 1–8, Apr. 2002. Iowa State University TR#2-06.

M. Wand, G. Kiczales, and C. J. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *Transactions on Programming Languages and Systems*, 26(4):890–910, Sept. 2004.

APPENDIX

A. PROC ELABORATOR

```
;;; Elaborator

(define *globs* #f) ;; (id × boxed-val)*
(define *procs* #f) ;; (id × proc/prim)*
(define *advs* #f)  ;; (pc × adv)

;; values – val ::= constant — procedure
(define-struct procV [ids body]) ;; PROC id* exp

(define _init-val_ 0) ;; val

;; location LOC ::= ref val (ie. box)

(define (lookup-glob i) ;; id → loc
  (let ([i+b (assq i *globs*)])
    (if i+b
        (cadr i+b)
        (error 'glob "not found: ~a" i))))

(define (lookup-proc i) ;; id → proc
  (let ([i+p (assq i *procs*)])
    (if i+p
        (cadr i+p)
        (error 'proc "not found: ~a" i))))

(define (get-glob l) ;; loc → val
  (unbox l))

(define (set-glob l v) ;; (loc × val) → val
  (let ([ov (unbox l)])
    (set-box! l v)
    ov))

(define ((lift o) v* k) ;; (val* → val) → (val* × cont) → !val
  (apply k (o v*)))

(define (elab! prims i+d*) ;; ((id × !(val* × cont))* × (id × decl))* → unit
  (set! *globs* '())
  (set! *procs* prims)
  (set! *advs* '())
  (for-each (lambda (i+d)
              (let ([d (cdr i+d)]
                    [i (car i+d)])
                (cond [(procD? d) (set! *procs* `((,i ,(make-procV (procD-ids d)
                                                                   (procD-body d)))
                                                  . ,*procs*))]
                      [(globD? d) (set! *globs* `((,i ,(box _init-val_))
                                                  . ,*globs*))]
                      [(advD? d) (set! *advs* `((,(advD-pc d) . ,(advD-body d))
                                                . ,*advs*))]
                      [else (error 'elab "not a decl: ~a" d)])))
            i+d*)
  (set! step (adv-step *advs*)))
```

Figure 14: Proc Elaborator

Aspects and Modular Reasoning in Nonmonotonic Logic

Klaus Ostermann
Darmstadt University of Technology, Germany
ostermann@informatik.tu-darmstadt.de

ABSTRACT

Nonmonotonic logic is a branch of logic that has been developed to model situations with incomplete information. We argue that there is a connection between AOP and nonmonotonic logic which deserves further study. As a concrete technical contribution and "appetizer", we outline an AO semantics defined in default logic (a form of nonmonotonic logic), propose a definition of modular reasoning, and show that the default logic version of the language semantics admits modular reasoning whereas a conventional language semantics based on weaving does not.

1. INTRODUCTION

There has been a lot of debate in the aspect-oriented community on how aspects influence program understanding or reasoning about programs, in particular how aspects influence "modular reasoning" (e.g., [12, 8]) (although modular reasoning has never really been defined). Previous works have concentrated on restricting AO languages in order to ease modular reasoning (e.g., [9, 2]). In this paper, we investigate a different approach: Rather than restricting the language, we propose to use a different reasoning model, namely nonmonotonic reasoning (see [3] for an overview).

In classical (monotonic) logic, adding a piece of information to a knowledge base never reduces the set of its consequences. Intuitively, monotonicity indicates that learning a new piece of knowledge cannot reduce what was previously known. Nonmonotonic logics (the formal incarnations of nonmonotonic reasoning) have been developed to deal with incomplete and changing information. Nonmonotonic logic allows to revise conclusions if new knowledge arrives, and provides rigorous mechanisms for taking back conclusions that no longer fit to newly learned knowledge, and deriving new, alternative conclusions instead.

In this paper, we argue that there is a fruitful connection between nonmonotonic logic and aspects. Using nonmonotonic logic, it is possible to specify the semantics of an AO language with pointcuts and advice in a very direct and compositional way: no kind of weaving or other global operation needs to be build into the semantic definitions. The absense of any operations requiring global knowledge means that reasoning with local knowledge is also easier. To validate this claim, we propose a definition of modular reasoning and show that nonmonotonic logic restores the ability for modular reasoning, albeit at the cost of giving up monotonicity.

The rest of the paper is structured as follows. In the next section, we give a very short introduction to default logic. In Sec. 3, we give a semantics of an AO language with pointcut and advice based on default logic and compare it with a conventional AO language semantics based on weaving. In Sec. 4 we consider the problem of modular reasoning and discuss how nonmonotonic logic influences modular reasoning. Sec. 5 discusses variants of default logic that employ priorities, and how these variants can be used to model advice precedence rules. Sec. 6 discusses what has been achieved.

Sixth International Workshop on Foundations of Aspect-Oriented Languages (FOAL 2007), March 13, 2007, Vancouver, BC, Canada.

2. DEFAULT LOGIC

A typical example in nonmonotonic logic is that we know birds usually fly, and that Tweety is a bird, and hence conclude that Tweety flies - until we learn that Tweety is actually a penguin. Using default logic [22] - one particular variant of nonmonotonic logic - we can formalize this situation as follows:

$$\frac{bird(X) \ : \ flies(X)}{flies(X)}$$

This rule is a so-called *default*, and can be read as "If X is a bird, and if it is consistent to assume that X flies (that is, it cannot be concluded that X does *not* fly), then conclude that X flies". In general, a default δ has the form $\frac{\varphi : \psi_1,...,\psi_n}{\chi}$, where $\varphi, \psi_1, ..., \psi_n, \chi$ are predicate logic formulae, and $n > 0$. The formula φ is called *prerequisite*, the part to the right of the colon, $\psi_1, ..., \psi_n$ *justifications*, and the part below the bar, χ, is the *consequent*. A default is *applicable* to a deductively closed set of formulae E, if $\varphi \in E$ and $\neg\psi_1 \notin E, ..., \neg\psi_n \notin E$.

In general, the set of conclusions that we can draw from a knowledge base with defaults is not unique. For example, if we know that members of the green party typically do not like cars, and members of an automobile club usually like cars, and John is members of both green party and automobile club, then we can conclude both that John likes cars and that he does not like cars.

This seeming chaos is ordered by so-called *extensions* - possible world views based on the given defaults. Technically, an extension is a superset of the knowledge base that

$$\frac{\vec{a} = \mathit{ApplicableAdvice}(o, m)}{...o.m(\vec{v}) \hookrightarrow ...o.m[\vec{a}](\vec{v})} \quad \text{(WEAVE)}$$

$$\frac{\mathit{AdviceLookup}(a) = (\vec{x}, e)}{...o.m[a, \vec{a}](\vec{v}) \hookrightarrow ...e\left[{}^{o}/_{\mathbf{this}}, {}^{\vec{v}}/_{\vec{x}}, {}^{o.m[\vec{a}](\vec{v})}/_{\mathsf{proceed}}\right]} \quad \text{(ADVEXEC)}$$

$$\frac{\mathit{MethodLookup}(o, m) = (\vec{x}, e)}{...o.m[\emptyset](\vec{v}) \hookrightarrow ...e\left[{}^{o}/_{\mathbf{this}}, {}^{\vec{v}}/_{\vec{x}}\right]} \quad \text{(METHEXEC)}$$

Figure 1: AO language semantics in the style of Jagadeesan et al

is consistent and closed under deduction and application of defaults [22]. In the example, we would have two distinct extensions, in which John likes and does not like, respectively, cars. A large part of the theory of default logic is concerned with the existence and construction of extensions and the relations between different extensions.

Reiter's original definition of extensions is a non-constructive fixed point equation based on the above properties, but we give an equivalent operational definition based on [18, 3, 4]. For this purpose, we define a *default theory* to be a pair $T = (W, D)$ consisting of a set W of predicate logic formulae (sometimes called *background theory*) and a countable set of defaults D. The extensions of T are the deductive closures of all sets E that can be generated by the following non-deterministic algorithm[1]:

$E := W; A := \emptyset;$
while there is a default $\delta \notin A$ that is applicable to E {
  $E := E \cup \{\mathit{consequent}(\delta)\}; A := A \cup \{\delta\};$
}
if $\forall \delta \in A.E$ is consistent with all justifications of δ
  then return E else failure

The algorithm first uses applicable defaults in an arbitrary order to build a candidate for an extension. The consistency check in the last two lines then checks whether E is really an extension. In general, extensions are neither unique (due to the non-deterministic choice of the next default) nor need to exist at all (due to the consistency check).

It may look strange that every default is applied at most once in the algorithm. This is sufficient, because the rule about birds above is technically not a default but a *default schema* since it contains a free variable (namely X). Default schemata are implicitly interpreted to mean the set of defaults $\frac{\varphi\sigma : \psi_1\sigma, ..., \psi_n\sigma}{\chi\sigma}$ for all ground substitutions σ that assign values to all free variables in the schema. For example, if we have two birds Tweety and Trixy, then our default schema creates two separate defaults $\frac{\mathit{bird}(\mathit{Tweety}) \ : \ \mathit{flies}(\mathit{Tweety})}{\mathit{flies}(\mathit{Tweety})}$ and $\frac{\mathit{bird}(\mathit{Trixy}) \ : \ \mathit{flies}(\mathit{Trixy})}{\mathit{flies}(\mathit{Trixy})}$.

3. ASPECTS AND DEFAULT LOGIC

Usually, pointcuts are implemented by static or dynamic weaving (that is, code transformation) or by interception and dynamic lookup. This view is also reflected in most formal accounts of AOP languages. Let us consider an object-oriented language with "around" advice that can advise method calls. In Jagadesaan et al's calculus of aspect-oriented programs [11], the (small-step) operational semantics rules for method lookup look roughly as sketched in Fig. 1. We leave out many details that are irrelevant for the purpose of this paper. The "..." part in the transition rules stands for dynamic entities of the operational semantics, such as call stacks or heaps. We also refrain from showing all the other rules of a complete operational semantics, since the rules for method and advice execution are sufficient to illustrate our idea.

A method call $o.m(\vec{v})$ is executed by first looking up all advice that applies to a method call and sorting the advice in some order (inside the *ApplicableAdvice* function, whose definition is not shown here), and weaving the sorted list of advice $\vec{a}$ into the method call (WEAVE). This weaved method call is then executed by taking the first advice from the list, looking up the formal arguments and body of the first advice, substituting **this** and the formal parameters $\vec{x}$ by the receiver object and the actual parameter values, respectively, and substituting `proceed` by a method call that removes the first advice from the list of pending advice (ADVEXEC). If no advice is left, the original method body is executed (METHEXEC). In both cases, the lookup functions return a list with the names of the formal parameters and the advice/method body.

Let us now study how we could encode a similar AO language semantics using default logic. We propose rules as presented in Fig. 2. The meaning of a method call is now a bit different: Whereas in Fig. 1 an expression $o.m[\vec{a}](\vec{v})$ denotes a method call where the execution of all advice in $\vec{a}$ is *pending*, we now interpret it to mean a method call where all advice in $\vec{a}$ *have already been executed*. Hence we do not need a separate syntactic form $o.m(\vec{v})$ for method calls before weaving; rather, normal method calls are denoted as $o.m[\emptyset](\vec{v})$.

There are only two computation rules, (METH) and (ADV). Due to the different meaning of the advice list in method calls, (ADV) *adds* rather than removes the name of the executed advice to the method call that replaces `proceed`. There is no weaving rule anymore. Rather, the behavior of (METH) and (ADV) is controlled by the auxiliary predicates *NextAdvice* and *unadvised*, which are defined using defaults. If there is no information to the contrary, we assume that a method call is unadvised (UNADV). If however, there is some applicable advice a that has not yet been executed, and if it is consistent to assume that it is the next advice to execute, then we conclude that a will be the next advice (NEXTADV). Furthermore, a call with applicable advice is not unadvised (SOMEADV).

To avoid that two different advice are both simultaneously the next one, we implicitly assume the existence of the usual inference rules of equality, in particular $\frac{x \neq x'}{\neg(x = x')}$.

[1]Recall the definition of *applicable* on the previous page

$$\frac{\begin{array}{c}MethodLookup(o,m) = (\vec{x}, e)\\ unadvised(o,m,\vec{a})\end{array}}{...o.m[\vec{a}](\vec{v}) \hookrightarrow ...e\left[{}^{o}/_{\mathbf{this}}, {}^{\vec{v}}/_{\vec{x}}\right]} \quad \text{(Meth)}$$

$$\frac{\begin{array}{c}NextAdvice(o,m,\vec{a}) = a\\ AdviceLookup(a) = (\vec{x}, e)\end{array}}{...o.m[\vec{a}](\vec{v}) \hookrightarrow ...e\left[{}^{o}/_{\mathbf{this}}, {}^{\vec{v}}/_{\vec{x}}, {}^{o.m[a,\vec{a}](\vec{v})}/_{\mathsf{proceed}}\right]} \quad \text{(Adv)}$$

$$\frac{true \ : \ unadvised(o,m,\vec{a})}{unadvised(o,m,\vec{a})} \quad \text{(Unadv)}$$

$$\frac{a \in ApplicableAdvice(o,m) \wedge a \notin \vec{a} \ : \ NextAdvice(o,m,\vec{a}) = a}{NextAdvice(o,m,\vec{a}) = a} \quad \text{(NextAdv)}$$

$$\frac{a \in ApplicableAdvice(o,m) \wedge a \notin \vec{a}}{\neg unadvised(o,m,\vec{a})} \quad \text{(SomeAdv)}$$

Figure 2: AO language semantics using default logic

To appreciate the difference between default and classical logic, assume for a moment that the colons in Fig. 2 would be replaced by conjunction operators (i.e., we would use classical rules). In this case, we could never prove a goal of the form $unadvised(o,m,\vec{a})$ or $NextAdvice(o,m,\vec{a}) = a$ because the same goal that we want to prove also appears in the premise of its rule. Hence the semantics would be useless. Similarly, if we would just remove the justifications, the semantics would be useless because we could prove $unadvised(o,m,\vec{a})$ for arbitrary o, m, and $\vec{a}$.

Now, the question arises whether the default theory in Fig. 2 has any extensions, and if any, what they look like. Luckily, all default rules in Fig. 2 are so-called *normal defaults*, meaning that the justification is the same as the consequent. Normal default theories are particularly well-behaved. Besides other important properties, normal default theories *always* possess extensions [22], which answers the first question.

Is there only a unique extension? No - in case more than one advice is applicable at some point, there is more than one extension, namely one for every possible advice execution order. This reflects the fact that there is no a-priori order among different overlapping advice. The difference to previous approaches is that we can now deal with this situation *within our reasoning framework*, and study the ambiguity in terms of extensions.

Let us now analyze informally to which degree the two language semantics agree with each other. If at most one pointcut applies at any joinpoint, the two semantics agree because in this case, there is only one unique extension in the default theory, which is the same theory that is generated by the conventional operational semantics. The semantics differ in how they treat shared joinpoints (more than one pointcut applies). In Fig. 1, the *ApplicableAdvice* lookup function orders all applicable advice in a specific order, whereas in Fig. 2 every potential execution order is represented by a different extension. We will later discuss how variants of default logic such as *prioritized default logic* [6] can be used to model global orders or ordering hints (such as *declare precedence* in AspectJ) on advice.

4. MODULAR REASONING

We will now attempt to give a semi-formal definition of modular reasoning. Reasoning can be performed with respect to a knowledge base, whereby we define a knowledge base as a set of logic formulae (or axioms) F in the case of classical reasoning, and as a default theory $T = (W, D)$ in the case of default reasoning. What can be concluded from the knowledge base is the deductive closure of F in the classical case, and the set of extensions of T in the default logic case.

We view the "partial evaluation" of the operational semantics rules with the current program as the knowledge base which we use to reason about the operational behavior of a program. By this we mean the set of rule instances where all meta-variables that refer to parts of the program are replaced by ground substitutions from the program. Recall that the operational semantics inference rules are actually rule schemata that stand for a set of rule instances, hence we can talk about the set of rule instances $ruleinstances(P)$ for a given program P[2]. For example, if our program contains an object *anObj* and a method *aMeth* of this object whose body returns **this**, then

$$\frac{Unadvised(anObj, aMeth, \emptyset)}{...anObj.aMeth[\emptyset]() \hookrightarrow ...anObj}$$

is a rule instance of (Meth).

Please note in this context that the lookup functions *AdviceLookup* etc. are very different from the *Unadvised* and *NextAdvice* predicates, in that the lookup functions have a fixed interpretation and can hence simply be unfolded in any rule instances, whereas the meaning of *Unadvised* and *NextAdvice* is *defined* in (Unadv) and (NextAdv), similarly to how $\hookrightarrow$ is defined in (Meth) and (Adv).

[2]We are cheating a bit because the program is usually (at least implicitly) a part of the derivation rules, e.g., derivation rules of the format $P \vdash e_1 \hookrightarrow e_2$. We have deliberately removed the program from the rules such that we can talk about preservation of rule instances with respect to program expansion.

Assuming some module structure in the underlying language, we say that a program P' is an *expansion* of P, if P' contains P but may contain additional modules. The definition of modular reasoning is as follows: *A language admits modular reasoning with respect to a set of rules, if, for all programs P and P' such that P' is an expansion of P, we have $ruleinstances(P) \subseteq ruleinstances(P')$.*

The rationale behind this definition is that the set of rule instances of a program can be considered as the knowledge base which we use to reason about the program. If we investigate a subpart of a program, then the knowledge base (i.e., the set of rule instances) should only grow when we investigate bigger parts of the program, but the knowledge base should never be invalidated by considering a larger part of the program.

Let us now consider how the language definitions in Fig. 1 and 2 perform with respect to this definition. The decisive rule in Fig. 1, which prevents modular reasoning, is (Weave). For example, we may have

$$...anObj.aMeth() \hookrightarrow ...anObj.aMeth[\emptyset]()$$

for some method $aMeth$ with zero parameters and object $anObj$ in a program P, but

$$...anObj.aMeth() \hookrightarrow ...anObj.aMeth[anAdv]()$$

in an expansion P' of P that adds an advice $anAdv$ for calls to $anObj.aMeth()$. Hence, modular reasoning (according to our definition), is not supported via this language definition.

The situation is different in Fig. 2, because there is no rule like (Weave) that needs global knowledge. To be concrete, assume a method call $anObj.aMeth[\emptyset]()$, where the body of $aMeth$ just returns **this**. Then we have a rule instance of the rule schema (Meth) which has the form $\frac{Unadvised(anObj,aMeth,\emptyset)}{...anObj.aMeth[\emptyset]() \hookrightarrow ...anObj}$. This rule instance is stable w.r.t. program expansion. If we consider again P' which adds advice $anAdv$, then this rule instance is still valid, but we get an additional rule instance of (SomeAdv), namely

$$\frac{true \wedge true}{\neg unadvised(anObj, aMeth, \emptyset)}$$

Hence, modular reasoning (according to our definition) is possible in the default logic version of the language semantics.

We believe that our approach also enables a form of modular verification in the sense of [14]: To determine whether an expansion of a program violates some property of the original program that holds in some extension, it is sufficient to check whether the set of assumptions A in our algorithm for computing extensions is consistent with the program expansion; it is not necessary to re-examine the whole program.

5. PRIORITIES

If two advice apply at some joinpoint, the question arises in which order the advice are to be executed. Languages like AspectJ leave the order unspecified (or use an arbitrary order such as lexicographic order of aspect names) by default, but enable the programmer to insert precedence rules into the program. Such mechanisms are very naturally supported in default logic, and we believe that the various results in this domain (see [6, 10, 23] for an overview) could be projected back to AO languages and lead to better priority specification mechanisms.

At this point, we will only consider two simple variants of default logic with priorities: PDL and PRDL [6] [3, Chap. 8]. In PDL, the priority information is given in the form of a *strict partial order* $<$ on the set of defaults. The set of extensions of a default theory in PDL is restricted to those extensions that respect $<$, i.e., the order of default application in the algorithm in Sec. 2 is compatible with $<$.

For the purpose of modelling constructs like `declare precedence` in AspectJ, PRDL is even more appropriate, because PRDL allows to model the priority information *within* the logic, rather than as an external partial order as in PDL. In PRDL, every default δ_i has a name d_i. It also introduces a special symbol $\prec$ acting on default names. $d_1 \prec d_2$ can be read as "give the default with name d_1 priority over the default with name d_2". A term $d_1 \prec d_2$ in PRDL is an ordinary formula that can be used both in the background theory W and in defaults D of a default theory $T = (W, D)$. So, an AspectJ precedence declaration `declare predence a1, a2` can be represented by adding $d \prec d'$ to W for every d, d' that is the name of a rule instance of (NextAdv) for `a1` and `a2`, respectively. Of course, in PRDL the notion of extension is refined to *priority extensions*, which respect the order hints in T. Note that PRDL is already a much more general model than AspectJ's `declare precedence`, because ordering hints can be given inside arbitrary logic formulaes. Since they may also be given inside defaults, it is even possible to model that different extensions use different priorities! With regard to aspect priority this would mean that the priority between two advice might depend on the choosen priority order between other advice.

One step further would be to consider *dynamic priorities*, which are well-known in nonmonotonic logic [7, 5]. We believe that these mechanisms could be directly used to design new advanced priority mechanisms for AO languages.

6. DISCUSSION

Using default logic, it is possible to define the semantics of an AO language in a compositional way, without using weaving or other kinds of global operations. This is not only interesting from the perspective of defining the language, but also from the perspective of reasoning about programs in the language, since a language semantics also influences how we reason about programs.

Our definition of modular reasoning seems to be a bit strange in that the whole concept of interfaces, which is usually a central notion of modularty, does not show up in any way. Hence, it is not clear whether our definition really fits to what people usually associate with the term "modular reasoning". If it does not, the author is happy to take any suggestion for a better name for this property. Another potential weakness of our definition is that it is possible to build up global (or at least non-local) information during execution, e.g., a list of dynamically deployed aspects that is propagated in the ... part of our reduction rules (such as in Lämmel's approach [15]). According to our definition, such an approach would still allow modular reasoning. This brings up the question of the difference (with respect to modular reasoning) between having to have global knowledge about the program, or knowledge about dynamic parameters that influence the execution and are propagated through the execution steps (such as aspect registries, heaps, or monads).

One may also argue that our definition of modular rea-

soning has been carefully worded to fit to our approach, and that what one really wants is monotonicity in *the set of conclusions* from a knowledge base and not so much monotonicity in the knowledge base itself. Indeed, our default logic version is nonmonotonic in this regard: A previously existing extension based on the assumption $unadvised(anObj, aMeth, \emptyset)$ can be invalidated by program expansion. This is what nonmonotonic logic is about, after all.

However, we still believe there is value in our approach because now we can deal with this nonmonotonicity in a reasoning framework that has been specifically developed for that very purpose. In Sec. 5 we have already hinted at how variants of nonmonotonic logic with priorities might fertilize AO language design. We believe that this is also true for other results from nonmonotonic logic. For example, the theory of default logic gives conditions under which extensions are unique or under which conclusions are contained in all extensions of a default theory [17, 13]. There is a systematic process to deal with changing belief sets [1]. There are mechanisms to keep track of the beliefs upon which we base our conclusions [16]. With our approach, we can now project these results from default logic back into the AO language domain. Connections between logic and programming have turned out to be quite fruitful in the past (Curry-Howard isomorphism!), and we hope that this connection between AOP and nonmonotonic logic is no exception.

In this work, we have concentrated on default logic. It would probably also be possible to define our language semantics in *autoepistemic logic* [21]. Autoepistemic logic introduces an operator L, where $L\phi$ is interpreted as 'I believe in ϕ'. Using this operator, our (UNADV) rule, for example, could be encoded as

$$\neg L \neg unadvised(o, m, \vec{a}) \rightarrow unadvised(o, m, \vec{a})$$

Since autoepistemic logic is intuitively based on introspection (rather than default rules), autoepistemic logic might provide another interesting reasoning framework to interpret AOP.

Another well-known approach in nonmonotonic logic is circumscription [19, 20]. We believe that circumscription could be useful to devise a model-theoretic interpretation of AOP. From the perspective of logic, most semantic accounts of AOP are proof-theoretic (including this one). Circumscription gives a model-theoretic interpretation of nonmonotonic logic by selecting minimal models from the space of models of a theory. We believe it would be possible to define a variant of our semantics where the *unadvised* and *NextAdvice* predicates are circumscribed (i.e., their meaning is minimized), rather than defining them via defaults. However, this is clearly a topic for future work.

7. REFERENCES

[1] C. E. Alchourrón, P. Gärdenfors, and D. Makinson. On the logic of theory change: Partial meet contraction and revision functions. *J. Symb. Log.*, 50(2):510–530, 1985.

[2] J. Aldrich. Open modules: Modular reasoning about advice. In *ECOOP'05*, Lecture Notes in Computer Science, pages 144–168. Springer, 2005.

[3] G. Antoniou. *Non-monotonic reasoning.* MIT Press, 1996.

[4] G. Antoniou. A tutorial on default logics. *ACM Comput. Surv.*, 31(4):337–359, 1999.

[5] G. Antoniou. Defeasible logic with dynamic priorities. In *Proceedings of the 15th Eureopean Conference on Artificial Intelligence, ECAI'2002*, pages 521–525. IOS Press, 2002.

[6] G. Brewka. Reasoning about priorities in default logic. In *Proceedings of the 12th national conference on Artificial intelligence (AAAI)*, pages 940–945. American Association for Artificial Intelligence, 1994.

[7] G. Brewka. Well-founded semantics for extended logic programs with dynamic preferences. *J. Artif. Intell. Res. (JAIR)*, 4:19–36, 1996.

[8] C. Clifton and G. T. Leavens. Obliviousness, modular reasoning, and the behavioral subtyping analogy. In *Workshop on Software engineering Properties of Languages for Aspect Technologies (SPLAT!) at AOSD 2003.*, 2003.

[9] D. S. Dantas and D. Walker. Harmless advice. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'06)*, pages 383–396. ACM, 2006.

[10] P. M. Dung and T. C. Son. An argument-based approach to reasoning with specificity. *Artif. Intell.*, 133(1-2):35–85, 2001.

[11] R. Jagadeesan, A. Jeffrey, and J. Riely. A calculus of untyped aspect-oriented programs. In *ECOOP 2003 - Object-Oriented Programming, 17th European Conference*, pages 54–73, 2003.

[12] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 49–58, New York, NY, USA, 2005. ACM Press.

[13] S. Kraus, D. J. Lehmann, and M. Magidor. Nonmonotonic reasoning, preferential models and cumulative logics. *Artif. Intell.*, 44(1-2):167–207, 1990.

[14] S. Krishnamurthi, K. Fisler, and M. Greenberg. Verifying aspect advice modularly. In *SIGSOFT FSE'04*, pages 137–146. ACM, 2004.

[15] R. Lämmel. A semantical approach to method-call interception. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 41–55, New York, NY, USA, 2002. ACM Press.

[16] W. Lukaszewicz. Considerations on default logic: an alternative approach. *Computational Intelligence*, 4:1–16, 1988.

[17] D. Makinson. General patterns in nonmonotonic reasoning. In *Handbook of logic in artificial intelligence and logic programming (vol. 3): nonmonotonic reasoning and uncertain reasoning*, pages 35–110, New York, NY, USA, 1994. Oxford University Press, Inc.

[18] W. Marek and M. Trusczynski. *Nonmonotonic Logic.* Springer, 1993.

[19] J. McCarthy. Circumscription—a form of non-monotonic reasoning. *Artificial Intelligence*, 13:27–39, 1980.

[20] J. McCarthy. Applications of circumscription to formalizing common sense knowledge. *Artificial Intelligence*, 28:89–116, 1986.

[21] R. C. Moore. Semantical considerations on nonmonotonic logic. *Artif. Intell.*, 25(1):75–94, 1985.

[22] R. Reiter. A logic for default reasoning. *Artif. Intell.*, 13(1-2):81–132, 1980.

[23] X. Zhao. Complexity of argument-based default reasoning with specificity. *AI Commun.*, 16(2):107–119, 2003.

Aspect-Oriented Programming with Type Classes

Martin Sulzmann
School of Computing,
National University of Singapore
S16 Level 5, 3 Science Drive 2,
Singapore 117543
sulzmann@comp.nus.edu.sg

Meng Wang
Oxford University Computing Laboratory,
Wolfson Building, Parks Road,
Oxford OX1 3QD, UK
meng.wang@comlab.ox.ac.uk

ABSTRACT

We consider the problem of adding aspects to a strongly typed language which supports type classes. We show that type classes as supported by the Glasgow Haskell Compiler can model an AOP style of programming via a simple syntax-directed transformation scheme where AOP programming idioms are mapped to type classes. The drawback of this approach is that we cannot easily advise functions in programs which carry type annotations. We sketch a more principled approach which is free of such problems by combining ideas from intentional type analysis with advanced overloading resolution strategies. Our results show that type-directed static weaving is closely related to type class resolution – the process of typing and translating type class programs.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Semantics*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Operational semantics*

General Terms

Languages, theory

Keywords

Type class resolution, type-directed weaving

1. INTRODUCTION

Aspect-oriented programming (AOP) is an emerging paradigm which supports the interception of events at run-time. The essential functionality provided by an aspect-oriented programming language is the ability to specify *what* computation to perform as well as *when* to perform the computation. A typical example is profiling where we may want to record the size of the function arguments (what) each time a certain function is called (when). In AOP terminology, what computation to perform is referred to as the *advice* and when to perform the advice is referred to as the *pointcut*. An *aspect* is a collection of advice and pointcuts belonging to a certain task such as profiling.

There are numerous works which study the semantics of aspect-oriented programming languages, for example consider [1, 13, 22, 24, 25, 27]. Some researchers have been looking into the connection between AOP and other paradigms such as generic programming [28]. To the best of our knowledge, we are the first to study the connection and combination between AOP and the concept of type classes, a type extension to support overloading (a.k.a. ad-hoc polymorphism) [23, 11], which is one of the most prominent features of Haskell [17].

In this paper, we make the following contributions:

- We introduce an AOP extension of Haskell, referred to as AOP Haskell, with type-directed pointcuts (Section 3.1).
- We define AOP Haskell by means of a syntax-directed translation scheme where AOP programming idioms are directly expressed in terms of type class constructs. Thus, typing and translation of AOP Haskell can be explained in terms of typing and translation of the resulting type class program (Section 5).

Our type class encoding of AOP critically relies on multi-parameter type classes and overlapping instances. Both features are not part of the Haskell 98 standard [17], but they are supported by the Glasgow Haskell Compiler (GHC) [4]. There are two problems.

Firstly, GHC's overlapping instances have never been formalized. Hence, it is difficult to make any precise claims regarding soundness of our GHC type class encoding of AOP. Secondly, the AOP to GHC type class translation scheme only works in case we do *not* advise programs which contain type annotations. Section 4.2 provides further details.

Despite these problems, we consider the encoding of AOP via GHC type classes a useful exercise. To encode AOP in the setting of a strongly typed language we need some form of type-safe cast. Type classes are known to have this capability and in our approach we achieve this by exploiting GHC's overlapping instances. Thus, we can establish that the concepts of type classes and aspects are closely related.

There are a number of works, for example consider [14, 12], which also use sophisticated type class tricks to model type safe casts in the setting of generic programming and strongly typed heterogeneous collections. These works may

Sixth International Workshop on Foundations of Aspect-Oriented Languages (FOAL 2007), March 13, 2007, Vancouver, BC, Canada.

even have a solution how to advise functions without having to change type annotations by using some more advanced type class "hackery". However, we do not plan to consider this avenue further. Ultimately, we seek for a more principled approach which allows us to study the combination of type classes and aspects without having to rely on the features of specific implementations such as GHC.

We are currently working on a foundational framework to integrate type classes and aspects. We briefly sketch this more principled approach in Section 6. The idea is to use Harper and Morrisett's intentional type analysis framework [6] for the translation of aspects and Stuckey and the first author's overloading framework [19] for the resolution of type classes programs with aspects.

We continue in Section 2 where we give an introduction to type classes. Section 3 gives an overview of the key ideas behind our approach of mapping AOP Haskell to GHC type classes. Section 4 discusses the shortcomings of this approach. We conclude in Section 7 where we also discuss related work.

2. BACKGROUND: TYPE CLASSES

Type classes [11, 23] provide for a powerful abstraction mechanism to deal with user-definable overloading also known as ad-hoc polymorphism. The basic idea behind type classes is simple. Class declarations allow one to group together related methods (overloaded functions). Instance declarations prove that a type is in the class, by providing appropriate definitions for the methods.

Here are some standard Haskell declarations.

```
class Eq a where (==)::a->a->Bool
instance Eq Int where (==) = primIntEq       -- (I1)
instance Eq a => Eq [a] where                -- (I2)
   (==) [] [] = True
   (==) (x:xs) (y:ys) = (x==y) && (xs==ys)   -- (L)
   (==) _ _ = False
```

The class declaration in the first line states that every type `a` in *type class* `Eq` has an equality function `==`. Instance (I1) shows that `Int` is in `Eq`. We assume that `primIntEq` is the (primitive) equality function among `Int`s. The common terminology is to express membership of a type in a type class via constraints. Hence, we say that the *type class constraint* `Eq Int` holds. Instance (I2) shows that `Eq [a]` from the instance *head* holds if `Eq a` in the instance *context* holds. Thus, we can describe an infinite family of (overloaded) equality functions.

We can extend the type class hierarchy by introducing new subclasses.

```
class Eq a => Ord a where (<)::a->a->Bool    -- (S1)
instance Ord Int where ...                   -- (I3)
instance Ord a => Ord [a] where ...          -- (I4)
```

The above class declaration introduces a new subclass `Ord` which inherits all methods of its superclass `Eq`. For brevity, we ignore the straightforward instance bodies.

In the standard type class translation approach we represent each type class via a dictionary [23, 5]. These dictionaries hold the actual method definitions. Each superclass is part of its (direct) subclass dictionary. Instance declarations imply dictionary constructing functions and (super) class declarations imply dictionary extracting functions. The dictionary translation of the above declarations is given in Figure 1.

```
type DictEq a = (a->a->Bool)
instI1 ::  DictEq Int
instI1 = primIntEq
instI2 ::  DictEq a -> DictEq [a]
instI2 dEqa =
 let eq [] [] = True
     eq (x:xs) (y:ys) = (dEqa x y) &&
                        (instI2 dEqa xs ys)
     eq _ _ = False
 in eq
type DictOrd a = (DictEq a, a->a->Bool)
superS1 ::  DictOrd a -> DictEq a
superS1 = fst
instI3 ::  DictOrd Int
instI3 = ...
instI4 ::  DictOrd a -> DictOrd [a]
instI4 = ...
```

Figure 1: Dictionary-Passing Translation

Notice how the occurrences of `==` on line (L) have been replaced by some appropriate dictionary values. For example, in the source program the expression `xs == ys` gives rise to the type class constraint `Eq [a]`. In the target program, the dictionary `instI2 dEqa` provides evidence for `Eq [a]` where `dEqa` is the (turned into a function argument) dictionary for `Eq a` and `instI2` is the dictionary construction function belonging to instance (I2).

The actual translation of programs is tightly tied to type inference. When performing type inference, we reduce type class constraints with respect to the set of superclass and instance declarations. This process is known as *type class resolution* (also known as context reduction). For example, assume some program text gives rise to the constraint `Eq [[[a]]]`. We reduce `Eq [[a]]` to `Eq a` via (reverse) application of instance (I2). Effectively, this tells us that given a dictionary `d` for `Eq a`, we can build the dictionary for `Eq [[a]]` by applying `instI2` twice. That is, `instI2 (instI2 d)` is the demanded dictionary for `Eq [[a]]`. Notice that given the dictionary `d'` for `Ord a`, we can build the alternative dictionary `instI2 (instI2 (superS1 d'))` for `Eq [[a]]`.

In the above, we only use single-parameter type classes. Other additional type class features include functional dependency [9], constructor [8] and multi–parameter [10] type classes. For the translation of AOP Haskell to Haskell we will use multi-parameter type classes and overlapping instances, yet another type class feature, as supported by GHC [4].

3. THE KEY IDEAS

To explain our idea of how to mimic AOP via GHC type classes, we first introduce an AOP extension of Haskell, referred to as AOP Haskell, and consider some example programs in AOP Haskell.

3.1 AOP Haskell

AOP Haskell extends the Haskell syntax [17] by supporting top-level aspect definitions of the form

```
N@advice #f1,...,fn# :: (C => t) = e
```

```
import List(sort)

insert x [] = [x]
insert x (y:ys)
  | x <= y    = x:y:ys
  | otherwise = y :  insert x ys

insertionSort [] = []
insertionSort xs =
  insert (head xs) (insertionSort (tail xs))

-- sortedness aspect
N1@advice #insert# ::  Ord a => a -> [a] -> [a] =
  \x -> \ys ->
     let zs = proceed x ys
     in if (isSorted ys) && (isSorted zs)
        then zs else error "Bug"
  where
     isSorted xs = (sort xs) == xs
-- efficiency aspect
N2@advice #insert# ::  Int -> [Int] -> [Int] =
  \x -> \ys ->
     if x == 0 then x:ys
     else proceed x ys
```

Figure 2: AOP Haskell Example

where `N` is a distinct label attached to each advice and the pointcut `f1`,...,`fn` refers to a set of (possibly overloaded) functions. Commonly, we refer to `fi`'s as *joinpoints*. Notice that our pointcuts are type-directed. Each pointcut has a type annotation `C => t` which follows the Haskell syntax. We refer to `C => t` as the *pointcut type*. We will apply the advice if the type of a joinpoint `fi` is an instance of `t` such that constraints `C` are satisfied. The advice body `e` follows the Haskell syntax for expressions with the addition of a new keyword `proceed` to indicate continuation of the normal evaluation process. We only support "around" advice which is sufficient to represent "before" and "after" advice.

In Figure 2, we give an example program. In the top part, we provide the implementation of an insertion sort algorithm where elements are sorted in non-decreasing order. At some stage during the implementation, we decide to add some security and optimization aspects to our implementation. We want to ensure that each call to `insert` takes a sorted list as an input argument and returns a sorted list as the result.

In our AOP Haskell extension, we can guarantee this property via the first aspect definition in Figure 2. We make use of the (trusted) library function `sort` which sorts a list of values. The `sort` functions assumes the overloaded comparison operator `<=` which is part of the `Ord` class. Hence, we find the pointcut type `Ord a=>[a]->[a]->[a]`. The keyword `proceed` indicates to continue with the normal evaluation. That is, we continue with the call `insert x ys`. The second aspect definition provides for a more efficient implementation in case we call `insert` on list of `Int`s. We assume that only non-negative numbers are sorted which implies that `0` is the smallest element appearing in a list of `Int`s. Hence, if `0` is the first element it suffices to cons `0` to the input list. Notice there is an overlap among the pointcut types for `insert`. In case we call `insert` on list of `Int`s we apply both advice bodies in no specific order unless otherwise stated. For all other cases, we only apply the first advice.

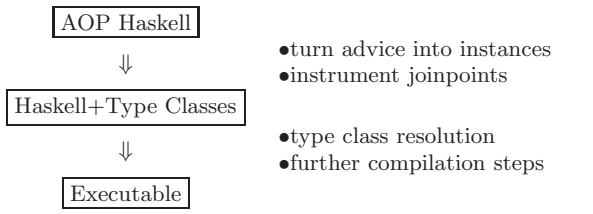

Figure 3: AOP Haskell Typing and Translation Scheme

Because AOP Haskell extends Haskell, we can naturally refer to overloaded functions in advice bodies. See the first advice body where our use of `sort` gives rise to the `Ord a` constraint. Also note the use of the (overloaded) equality operator `==` whose type is `Eq a => a -> a -> a`. In Haskell, the `Eq` class is a superclass of `Ord`. Hence, there is no need to mention the `Eq` class in the pointcut type of the advice definition.

3.2 Typing and Translating AOP Haskell with GHC Type Classes

Our goal is to embed AOP Haskell into Haskell by making use of Haskell's rich type system. Specifically, we use GHC with two extensions (multi-parameter type classes and overlapping instances). We give a transformation scheme where typing and translation of the *source* AOP Haskell program is described by the resulting *target* Haskell program.

The challenge we face is how to intercept calls to joinpoints and re-direct the control flow to the advice bodies. In AOP terminology, this process is known as aspect weaving. Weaving can either be performed dynamically or statically. Dynamic weaving is the more flexible approach. For example, aspects can be added and removed at run-time. For AOP Haskell, we employ static weaving which is more restrictive but allows us to give stronger static guarantees about programs such as type inference and type soundness.

Our key insight is that type-directed static weaving can be phrased in terms of type classes based on the following principles:

- We employ type class instances to represent advice.
- We use a syntactic pre-processor to instrument joinpoints with calls to overloaded "weaving" function.
- We explain type-directed static weaving as type class resolution. Type class resolution refers to the process of reducing type class constraints with respect to the set of instance declarations.

Figure 3 summarizes our approach of typing and translating AOP Haskell. In Figure 4, we apply the transformation scheme to the AOP Haskell program from Figure 2. We use here type classes as supported by GHC.

Let us take a closer look at how this transformation scheme works. First, we introduce a two-parameter type class `Advice` which comes with a method `joinpoint`. Each call to `insert` is replaced by

```
insert x [] = [x]
insert x (y:ys)
 | x <= y   = x:y:ys
 | otherwise=
 y :  (joinpoint N1 (joinpoint N2 insert)) x ys --(1)

insertionSort [] = []
insertionSort xs =
 (joinpoint N1 (joinpoint N2 insert)) --(2)
 (head xs) (insertionSort (tail xs))

-- translation of advice
class Advice n t where
 joinpoint ::  n -> t -> t
 joinpoint _ = id      -- default
data N1 = N1
instance Ord a => Advice N1 (a->[a]->[a]) where -- (I1)
joinpoint N1 insert =
 \x -> \ys -> let zs = insert x ys
              in if (isSorted ys) && (isSorted zs)
                 then zs else error "Bug"
 where
    isSorted xs = (sort xs) == xs
instance Advice N1 a -- (I1') default case

data N2 = N2
instance Advice N2 (Int->[Int]->[Int]) where -- (I2)
 joinpoint N2 insert = \x -> \ys ->
              if x == 0 then x:ys
              else insert x ys
instance Advice N2 a -- (I2') default case
```

Figure 4: GHC Haskell Translation of Figure 2

```
joinpoint N1 (joinpoint N2 insert)
```

We assume here the following order among advice: `N2` $\leq$ `N1`. That is, we first apply the advice `N1` before applying advice `N2`. This transformation step requires to traverse the abstract syntax tree and can be automated by pre-processing tools such as Template Haskell [18].

Next, each piece of advice is turned into an instance declaration where the type parameter `n` of the `Advice` class is set to the singleton type of the advice and type parameter `t` is set to the pointcut type. In case the pointcut type is of the form `C => ...`, we set the instance context to `C`. See the translation of advice `N1`. In the instance body, we simply copy the advice body where we replace `proceed` by the name of the advised function. Additionally, for each advice `N` we introduce `instance Advice N a` where the body of this instance is set to the default case as specified in the class declaration. The reader will notice that for each advice we create two "overlapping" instances. For example, the head `Advice N1 (a->[a]->[a])` of instance (I1) and the head `Advice N1 a` of the default instance (I1') overlap because the type components are unifiable (after renaming the `a` in `Advice N1 a` with a fresh variable `b`). Therefore, we can potentially use either of the two instances to resolve a type class constraint which may yield to two different results. However, GHC will postpone resolution of type classes until we can unambiguously choose an instance. We say that GHC implements a "lazy" and "best-fit" type class resolution strategy.

The actual (static) weaving of the program is performed by the type class resolution mechanism. GHC will infer the following types for the transformed program.

```
insert :: forall a.
          (Advice N1 (a -> [a] -> [a]),
           Advice N2 (a -> [a] -> [a]),
           Ord a) => a -> [a] -> [a]

insertionSort :: forall a.
          (Advice N1 (a -> [a] -> [a]),
           Advice N2 (a -> [a] -> [a]),
           Ord a) => a -> [a] -> [a]
```

Each `Advice` type class constraint results from a call to `joinpoint`. GHC's "lazy" type class resolution strategy does not resolve `Advice N1 (a -> [a] -> [a])` because we could either apply instance (I1) or the default instance (I1') which may yield to an ambiguous result. However, if we use `insert` or `insertionSort` in a specific monomorphic context we can resolve "unambiguously" the above constraints.

Let us assume we apply `insertionSort` to a list of `Int`s. Then, we need to resolve the constraints

```
(Advice N1 (Int -> [Int] -> [Int]),
 Advice N2 (Int -> [Int] -> [Int]), Ord Int)
```

GHC's "best-fit" strategy resolves `Advice N1 (Int -> [Int] -> [Int])` via instance (I1), `Advice N2 (Int -> [Int] -> [Int])` via instance (I2) and `Ord Int` is resolved using a predefined instance from the Haskell Prelude [17]. Effectively, this means that at locations (1) and (2) in the above program text, we intercept the calls to `insert` by first applying the body of instance (I1) followed by applying the body of instance (I2)

In case, we apply `insertionSort` to a list of `Bool`s, we need to resolve the constraints

```
(Advice N1 (Bool -> [Bool] -> [Bool]),
 Advice N2 (Bool -> [Bool] -> [Bool]), Ord Bool)
```

The instance (I1) is still the best-fit for `Advice N1 (Bool -> [Bool] -> [Bool])`. However, instead of instance (I2) we apply the default case to resolve `Advice N2 (Bool -> [Bool] -> [Bool])`. Hence, at locations (1) and (2) we apply the body of instance (I1) followed by the body of the default instance for advice (I2). `Ord Bool` is resolved using a pre-defined instance from the Haskell Prelude.

4. DISCUSSION

The transformation from AOP Haskell to Haskell using GHC type classes is simple and only requires a syntactic transformation of programs. In Section 5, we give the details plus further examples. We also show how to statically detect useless advice. Unfortunately, our AOP to type class transformation scheme suffers from the following problems:

1. Aspects must be pure, i.e. free of side-effects.
2. (a) Advising type annotated requires to rewrite annotations. (b) Rewriting of type annotations of polymorphic recursive functions is impossible.
3. The transformation scheme relies on multi-parameter type classes and overlapping instances extensions which are not part the Haskell 98 standard. But they are supported by GHC.

We will discuss each of the above three issues in turn.

4.1 Aspects Must be Pure

In Haskell the effect of a program is manifested in its (monadic) type. For example, a program which reads and writes to standard I/O will have type `IO ()` where `IO` belongs to the monad class. Hence, based on the syntax-directed transformation scheme described so far, aspects cannot make pure functions do I/O (for example, to do logging) or modify state (for example, to add memorization). We would need to "semantically rewrite" the program during the transformation, by for example changing an advised function of type `t` to a function of type `IO t` in case of an aspect with effect `IO`.

A possible systematic solution is to monadify programs [2] and use (state) monad transformers [15]. Another alternative is to use `unsafePerformIO` which obviously breaks type safety. We consider the issue of impure aspects as orthogonal to our work is which about establishing a connection between AOP and the concept of type classes. We plan to take a look at impure aspects in future work.

4.2 Advising Type Annotated Programs

Let us assume we provide explicit type annotations to the functions in Figure 2.

```
insert :: Ord a => a -> [a] -> [a]
insertionSort :: Ord a => [a] -> [a]
```

The trouble is that if we keep `insert`'s annotation in the resulting target program, we find some unexpected behavior. GHC's type class resolution mechanism will "eagerly" resolve the constraints

```
Advice N1 (a -> [a] -> [a]),
Advice N2 (a -> [a] -> [a])
```

arising from

```
joinpoint N1 (joinpoint N2 insert)
```

by applying instance (I1) on `Advice N1 (a -> [a] -> [a])` and applying the default instance (I2') on `Advice N2 (a -> [a] -> [a])`. Hence, will never apply the advise `N2`, even if we call insert on list of `Int`s.

The conclusion is that we must either remove type annotations in the target program, or appropriately rewrite them during the translation process. For example, in the translation we must rewrite `insert`'s annotation to

```
insert :: (Advice N1 (a -> [a] -> [a]),
           Advice N2 (a -> [a] -> [a]), Ord a) =>
           a -> [a] -> [a]
```

The need for rewriting type annotations complicates our simple AOP Haskell to Haskell transformations. In fact, in case of polymorphic recursive functions, which demand type annotations to guarantee decidable type inference [7], we are unable to appropriately the type annotation.

Let us consider a (contrived) program to explain this point in more detail. In Figure 5, function `f` makes use of polymorphic recursion in the second clause. We call `f` on list of lists whereas the argument is only a list. Function `f` will not terminate on any argument other than the empty list. Notice that the lists in the recursive call are getting "deeper' and "deeper". The advice definition allows us to intercept all calls to `f` on list of list of `Bool`s to ensure termination for at least some values.

```
f ::  [a] -> Bool
f [] = True
f (x:xs) = f [xs]

N@advice ♯f♯ ::  [[Bool]] -> Bool = \x -> False
```

Figure 5: Advising Polymorphic Recursive Functions

To translate the above AOP Haskell program to Haskell with GHC type classes we cannot omit `f`'s type annotation because `f` is a polymorphic recursive function. Our only hope is to rewrite `f`'s type annotation. For example, consider the attempt.

```
f :: Advice N a => [a] -> Bool
f [] = True
f (x:xs) = (joinpoint N f) [xs]
```

The call to `f` in the function body gives rise to `Advice N [a]` whereas the annotation only supplies `Advice N a`. Therefore, the GHC type checker will fail. Any similar "rewrite" attempt will lead to the same result (failure).

A closer analysis shows that the problem we face is due to the way type classes are implemented in GHC via the dictionary-passing scheme [5]. In fact, almost all Haskell implementations use the dictionary-passing scheme. Hence, the following observation applies to pretty much all Haskell implementations. In the dictionary-passing scheme, each each type class is represented by a dictionary containing the method definitions. In our case, dictionaries represent the advice which will be applied to a joinpoint. Let us assume we initially call `f` with a list of `Bool`s. Then, the default advice applies and we proceed with `f`'s evaluation. Subsequently, we will call `f` on a list of list of `Bool`s. Recall that `f` is a polymorphic recursive function. Now, we wish that the advice `N` applies to terminate the evaluation with result `False`. The problem becomes now clear. The initial advice (i.e. dictionary) supplied will need to be changed during the evaluation of function `f` We cannot naturally program this behavior via GHC type classes.

4.3 Transformation Requires Type Class Extensions

To encode AOP in the setting of a strongly typed language we need some form of type-safe cast. Multi-parameter type classes are not essential but GHC style overlapping instances are essential. However, GHC style overlapping instances are heavily debated and still lack a formal description.

4.4 Short Summary

The AOP Haskell to Haskell transformation scheme based on GHC type classes is simple. The problem is that we cannot advise programs which contain type annotations, unless we manually rewrite type annotations. This is impossible in case we advice polymorphic recursive functions. The source of the problem is the dictionary-passing scheme which underlies the translation of type classes in GHC.

A less well known fact is that there exist alternative type class translation proposals based on a type-passing translation scheme [21, 6]. The key insight is that if we employ a type-passing scheme for the translation of aspects we

can easily solve the problems of the GHC based translation scheme. We sketch such an approach in Section 6. First, we provide the details of mapping AOP Haskell to Haskell using GHC type classes.

5. AOP GHC HASKELL

We consider an extension of GHC with top-level aspect definitions of the form

```
N@advice #f1,...,fn# :: (C => t) = e
```

We omit to give the syntactic description of Haskell programs which can be found elsewhere [17]. We assume that type annotation `C => t` and expression `e` follow the Haskell syntax (with the addition of a new keyword `proceed` which may appear in `e`). We assume that symbols `f1`,...,`fn` refer to the names of (top-level) functions and methods (i.e. overloaded functions). See also Section 3.1.

As motivated in Section 4.2, we impose the following condition on the AOP extension of GHC.

Definition 1 (AOP GHC Haskell Restriction). *We demand that inside the lexical scope of a type annotation, advice or instance declaration there are no joinpoints.*

Notice that instance declarations "act" like type annotations. In the upcoming translation scheme we will translate advice declarations to instance declarations. Hence, joinpoints cannot be enclosed by advice and instance declarations either.

Next, we formalize the AOP to type class transformation scheme. We will conclude this section by providing a number of programs written in AOP GHC Haskell.

5.1 Type Class-Based Transformation Scheme

Based on the discussion in Section 3.2, our transformation scheme proceeds as follows.

Definition 2 (AOP to GHC Transformation). *Let p be an AOP Haskell program. We perform the following transformation steps on p to obtain the program p'.*

Advice class: *We add the class declaration*

```
class Advice n t where
       joinpoint :: n -> t -> t
       joinpoint _ = id  -- default case
```

Advice bodies: *Each AOP Haskell statement*

```
N@advice #f1,...,fn# :: C => t = e
```

is replaced by

```
data N = N
instance C => Advice N t where
  joinpoint _ proceed = e
instance Advice N a -- resolves to default case
```

Joinpoints: *For each function* `f` *and for all advice* `N1`*, ...,* `Nm` *where* `f` *appears in their pointcut we replace* `f` *by*

```
joinpoint N1 (... (joinpoint Nm f)...)
```

being careful to avoid name conflicts in case of lambda-bound function names. We assume that the order among advice is as follows: `Nm` $\leq$ ... $\leq$ `N1`.

To compile the resulting program we rely on the following GHC extensions (compiler flags):

- `-fglasgow-exts`
- `-fallow-overlapping-instances`

The first flag is necessary because we use multi-parameter type classes. The second flag enables support for overlapping instances.

Claim 1. *Type soundness and type inference for AOP GHC Haskell are established via translation to GHC-style type classes.*

We take it for granted that GHC is type sound and type inference is correct. However, it is difficult to state any precise results given the complexity of Haskell and the GHC implementation.

In our current type class encoding of AOP we do not check whether advice definitions have any effect on programs. For example, consider

```
f :: Int
f = 1

N@advice #f# :: Bool = True
```

where the advice definition `N` is clearly useless. We may want to reject such useless definitions by adding the following transformation step to Definition 2. The advice is useful if the program text resulting from the transformation step below is well-typed.

Useful Advice: Each AOP Haskell statement

```
N@advice #f1,...,fn# :: C => t = e
```

generates

```
eq :: a -> a -> a
eq = undefined
f1' :: C => t
f1' = undefined
f1'' = eq f1 f1'
...
fn' :: C => t
fn' = undefined
fn'' = eq fn fn''
```

in `p'` where `eq`, `f1'`,`f1''` ..., `fn'`, `fn''` are fresh identifiers.

We may be tempted to generate the following simpler program text.

```
fi' :: C => t
fi' = fi
```

This will work for the above program. But such a transformation scheme is too restrictive as the following example shows.

```
accF xs acc = accF (tail xs) (head xs :  acc)
reverse ::  [a] -> [a] -> [a]
reverse xs = accF xs []
append ::  [a] -> [a] -> [a]
append xs ys = accF xs ys

N@advice ♯accF♯ ::  [a] -> [a] -> [a] =
   \xs -> \acc -> case xs of
                    [] -> acc
                    _ -> proceed xs acc
```

Figure 6: Advising Accumulator Recursive Functions

```
module CollectsLib where

class Collects c e | c -> e where
  insert ::  e -> c -> c
  test ::  e -> c -> Bool
  empty ::  c

instance Ord a => Collects [a] a where
  insert x [] = [x]
  insert x (y:ys)
    | x <= y = x:y:ys
    | otherwise = y :  (insert x ys)
  test x xs = elem x xs
  empty = []
```

Figure 7: Collection Library

```
g :: [a] -> Int

N@advice #g# :: [a] -> a = ...
```

The advice is clearly useful (in case `a` is `Int`. However, the program text

```
g' :: [a] -> a
g' = g
```

is ill-typed because the annotation is too polymorphic.

The idea behind the useful advice transformation step is to test whether the combination of type constraints from `f1` and `C => t` is consistent (i.e. well-typed). Then, the advice must be useful.

5.2 AOP GHC Haskell Examples

We take a look at a few AOP GHC Haskell example programs. We will omit the translation to (GHC) Haskell which can be found here [20]. We also discuss issues regarding the scope of pointcuts and how to deal with cases where the joinpoint is enclosed by an annotation.

Advising recursive functions. Our first example is given in Figure 6. We provide definitions of `append` and `reverse` in terms of the accumulator function `accF`. We deliberately left out the base case of function `accF`. In AOP GHC Haskell, we can catch the base case via the advice `N`. It

```
module Main where

import List(sort)
import CollectsLib

insertionSort [] = []
insertionSort xs =
  insert (head xs) (insertionSort (tail xs))

N1@advice ♯insert♯ ::  Ord a => a -> [a] -> [a] =
  \x -> \ys ->
     let zs = proceed x ys
     in if (isSorted ys) && (isSorted zs)
        then zs else error "Bug"
  where
     isSorted xs = (sort xs) == xs

N2@advice ♯insert♯ ::  Int -> [Int] -> [Int] =
  \x -> \ys -> if x == 0 then x:ys
               else proceed x ys
```

Figure 8: Advising Overloaded Functions

is safe here to give append and reverse type annotations, although, the joinpoint is then enclosed by a type annotation. The reason is that only one advice `N` applies here.

Advising overloaded functions. In our next example, we will show that we can even advise overloaded functions. We recast the example from Section 3.1 in terms of a library for collections. See Figures 7 and 8. We use the functional dependency declaration `Collects c e | c->e` to enforce that the collection type `c` uniquely determines the element type `e`. We use the same aspect definitions from earlier on to advise function `insertionSort` and the now overloaded function `insert`. As said, we only advise function names which are in the same scope as the pointcut. Hence, our transformation scheme in Definition 2 effectively translates the code in Figure 8 to the code shown in Figure 4. The code in Figure 7 remains unchanged.

Advising functions in instance declarations. If we wish to advise all calls to `insert` throughout the entire program, we will need to place the entire code into one single module. Let us assume we replace the statement `import CollectsLib` in Figure 8 by the code in Figure 7 (dropping the statement `module CollectsLib where` of course). Then, we face the problem of advising a function enclosed by a "type annotation". Recall that instance declarations act like type annotations and there is now a joinpoint `insert` within the body of the instance declaration in scope. Our automatic transformation scheme in Definition 2 will not work here. The resulting program may type check but we risk that the program will show some "unaspect"-like behavior. The (programmer-guided) solution is to manually rewrite the instance declaration during the transformation process which roughly yields the following result

```
...
instance (Advice N1 (a->[a]->[a]),
          Advice N2 (a->[a]->[a]),
          Ord a) => Collects [a] a where
```

```
N1@advice ♯f♯ ::  [Int] -> Int =
   \xs -> (head xs) + (proceed (tail xs))


N2@advice ♯head♯ ::  [Int] -> Int =
   \xs -> case xs of
            [] -> -1
            _ -> proceed xs
```

Figure 9: Advising functions in advice bodies

```
type T = [Int] -> Int
data N1 = N1
instance Advice N2 T => Advice N1 T where
  joinpoint N1 f =
    \ xs -> ((joinpoint N2 head) xs) + (f (tail xs))

data N2 = N2
instance Advice N2 T where
  joinpoint N2 head =
    \xs -> case xs of
             [] -> -1
             _ -> head xs
```

Figure 10: GHC Haskell Translation of Figure 9

```
 insert x [] = [x]
 insert x (y:ys)
  | x <= y    = x:y:ys
  | otherwise =
     y : ((joinpoint N2 (joinpoint N1 insert)) x ys)
...
```

To compile the transformed AOP GHC Haskell program with GHC, we will need to switch on the following additional compiler flag:

- `-fallow-undecidable-instances`

We would like to stress that type inference for the transformed program is decidable. The "decidable instance check" in GHC is simply conservative, hence, we need to force GHC to accept the program.

Advising functions in advice bodies. Given that we translate advice into instances, it should be clear that we can also advise functions in advice bodies if we are willing to "guide" the translation scheme. In Figure 9, we give such an example and its (manual) translation is given in Figure 10. We rely again on the "undecidable" instance extension in GHC.

The last example makes us clearly wish for a system where we do not have to perform any manual rewriting. Of course, we could automate the rewriting of annotations by integrating the translation scheme in Definition 2 with the GHC type inferencer. However, the problem remains that we are unable to advise polymorphic recursive functions. Recall the discussion in Section 4.2.

6. TOWARDS A FRAMEWORK FOR TYPE CLASSES AND ASPECTS

We are currently working on a core calculus to study type classes and aspects. The two key ingredients are (1) a type-directed translation scheme from a calculus with type classes and aspects to a variant of Harper and Morrisett's λ_i^{ML} calculus, and (2) a type inference scheme for type class and aspect resolution based on Stuckey and the first author's overloading framework.

We illustrate the key ideas behind this approach via a simple example. We consider parts of the earlier program in Figure 2.

```
import List(sort)
insert ::  Ord a => a -> [a] -> [a]
insert x [] = []
insert x (y:ys) =
  if x <=y then x:y:ys else y :  insert x ys
N1@advice #insert# ::  Ord a => a -> [a] -> [a] = ...
N2@advice #insert# ::  Int -> [Int] -> [Int] = ...
```

We leave out the `insertionSort` function and also omit the advice bodies for brevity. Notice that `insert` carries a type annotation. Earlier we saw that in AOP GHC Haskell we cannot easily advise type annotated functions unless we rewrite type annotations.

We can entirely avoid rewriting of type annotations by switching to a type-passing translation scheme for the translation of advise. Type classes can be translated using the standard dictionary-passing scheme. Here is the translation of the above program.

```
insert = Λ a.  λ d:DictOrd a.  λ x:a.  λ xs:[a].
  case xs of
    [] → [x]
    (y:ys) →
       if (d (<=)) x y then x:y:ys -- (1)
       else y :  (
            (joinpoint N1 (a->[a]->[a]) d -- (2)
            ((joinpoint N2 (a->[a]->[a])) (insert a d)))
            x ys)

joinpoint = Λ n.  Λ a.
  typecase (n,a) of
    (N1,a->[a]->[a]) → λ d:DictOrd a.  ...  -- (3)
    (N1,_) → ...
    (N2,Int->[Int]->[Int]) → ...
    (N2,_) → ...
```

In the translation, function `insert` expects an additional type and dictionary argument. Dictionaries serve the usual purpose. For example, at location (1) the program text `d (<=)` selects the greater-or-equal method from the dictionary of the `Ord` class. The novelty are the additional type arguments which we use for selecting the appropriate advice. See locations (2) and (3).

Similarly to AOP GHC Haskell, we instrument joinpoints with calls to the weaving function `joinpoint`. The difference to AOP GHC Haskell is that we rely on run-time type information and use type case, which is part of the λ_i^{ML} calculus, to select the advice. For example, the advice `N1` is selected using the first two (type) arguments. The (translated) advice body assumes a third (dictionary) argument, see location (2), because we make use of the `Ord` class in

the advice body. Recall the full definition of advice `N1` in Figure 2. At the call site of the weaving function, we must of course supply the necessary arguments. See location (1).

The translation of programs is driving by type inference. To obtain a type inference algorithm, we map type class and advice declarations to Constraint Handling Rules (CHRs) [3]. In [19], Stuckey and the first author introduced an overloading framework where CHRs are used to reason about type class relations. By mapping type classes to CHRs, we can concisely reason about their type inference properties. CHRs have a simple operational semantics and thus we obtain a type inference algorithm.

For example, the type class instance

```
instance Ord a => Ord [a]
```

is mapped to the CHR

```
Ord [a] <==> Ord a
```

Logically, the symbol `<==>` stands for bi-implication. Operationally, the symbol `<==>` indicates a rewrite relation among constraints (from left to right). In contrast to Prolog, we only perform matching but *not* unification when rewriting constraints. Rewriting of constraints with respect to instances is also known as "context reduction" in the type class literature.

Here, we employ CHRs to reason about advice declarations. The advice declarations of our running example translate to the CHRs

```
Advice N1 (a->[a]->[a]) <==> Ord a
Advice N1 b <==> b /=(a->[a]->[a]) | True
Advice N2 (Int->[Int]->[Int]) <==> True
Advice N2 b <==> b /= (Int->[Int]->[Int]) | True
```

The first and third CHR result from the advice declarations `N1` and `N2` whereas the second and fourth CHR cover the "default" cases. Notice that the second and fourth CHR contain guard constraints which impose additional conditions under which a CHR can fire. For example, the second CHR will only rewrite `Advice N1 b` to `True` (the always true constraint), if `b` is not an instance of `(a->[a]->[a])`. The idea is that via guard constraints we can guarantee that the CHR representing the advice declaration and the default case do not overlap.

We will use the CHRs resulting from type class and advice declarations to solve (i.e. rewrite) constraints arising during type inference. In the translation, `y : insert x ys` is translated to

```
y :  ((joinpoint N1 (a->[a]->[a]) d
((joinpoint N2 (a->[a]->[a])) (insert a d)))
x ys)
```

and this program text gives rise to

```
Ord a, Advice N1 (a->[a]->[a]), Advice N2 (a->[a]->[a])
```

The first constraint arise from the call to `insert`. The second and third constraint arise from the instrumentation. The surrounding program text carries the type annotation `Ord a => a->[a]->[a]`. To check that the type annotation is correct we must perform a *subsumption* test among types which boils down to a *entailment* test among constraints. For our example, we must verify that `Ord a, Advice N1 (a->[a]->[a]), Advice N2 (a->[a]->[a])` follow from `Ord a`. In general, we must verify that the constraints C_1 from the annotation *entail* the constraints C_2 arising from the program text of that annotation, written $C_1 \supset C_2$.

The entailment check is fairly standard for type classes. We exhaustively rewrite C_2 with respect to the instances (i.e. CHRs) until we reach the from C_1, written $C_2 \rightarrowtail^* C_1$. In the presence of advice, this entailment checking strategy will not work anymore because none of the above CHRs applies to `Advice N2 (a->[a]->[a])`. But if we interpret the above CHRs as some logical formula P we find that `Advice N2 (a->[a]->[a])` is a logical consequence of any first-order model of P and `Ord a`. We can verify this fact via a simple case analysis. If `a` equals to `Int` then `Advice N2 (a->[a]->[a])` is equivalent to `True` because of the third CHR. Otherwise, via the fourth CHR we can conclude that `Advice N2 (a->[a]->[a])` is yet again equivalent to `True`. Hence, `Advice N2 (a->[a]->[a])` is a logical consequence of P.

Our idea is to extend the standard rewriting-based approach to check entailment by incorporating a case analysis. Of course, we need to guarantee that the extended entailment check remains decidable for Haskell 98 type classes and advice declarations making use of such type classes.

7. CONCLUSION AND RELATED WORK

There is a large amount of works on the semantics of aspect-oriented programming languages, for example consider [1, 13, 22, 24, 25, 27] and the references therein. There have been only a few works [24, 1, 16] which aim to integrate AOP into ML style languages. We yet have to work out the exact connections to these works. For instance, the work described in [1] supports first-class pointcuts and dynamic weaving whereas our pointcuts are second class and we employ static weaving. None of the previous works we are aware of considers the integration of AOP and type classes. In some previous work, the second author [27, 26] gives a a static weaving scheme for a strongly typed functional AOP language via a type-directed translation process. In [27] (Section 6), the authors acknowledge that their type-directed translation scheme for advice is inspired by the dictionary-passing translation for type classes. But they believe that aspects and type classes substantially differ when it comes to typing and translation. In this paper, we confirm that there is a fairly tight connection between aspects and type classes. The system described in [27, 26] does not assume type annotations and therefore we can express all examples from [27, 26] in terms AOP GHC Haskell (Section 5).

The main result of our work is that static weaving for strongly typed languages bears a strong resembles to type class resolution – the process of typing and translating type class programs. We could show that GHC type classes as of today can provide for a light-weight AOP extension of Haskell (Section 5). We use GHC style overlapping instances to encode a form of type-safe which is is used to advice functions based on their types. The approach has a number of problems. For example, we cannot easily advise functions in programs with type annotations.

We are in the process of formalizing a more principled approach to integrate type classes and aspects (Section 6). We expect to report results in the near future. Further future work includes the study of effect-full advice which we can represent via monads in Haskell. We also want to consider more complex pointcuts.

Acknowledgments

We thank Andrew Black, Ralf Lämmel, and referees for AOSD'07 and FOAL'07 for their helpful comments on previous versions of this paper.

8. REFERENCES

[1] D. S. Dantas, D. Walker, G. Washburn, and S. Weirich. PolyAML: a polymorphic aspect-oriented functional programming language. In *Proc. of ICFP'05*, pages 306–319. ACM Press, 2005.

[2] M. Erwig and D. Ren. Monadification of functional programs. *Sci. Comput. Program.*, 52(1-3):101–129, 2004.

[3] T. Frühwirth. Constraint handling rules. In *Constraint Programming: Basics and Trends*, LNCS. Springer-Verlag, 1995.

[4] Glasgow haskell compiler home page. http://www.haskell.org/ghc/.

[5] C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, 1996.

[6] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Proc. of POPL'95*, pages 130–141. ACM Press, 1995.

[7] Fritz Henglein. Type inference with polymorphic recursion. *Transactions on Programming Languages and Systems*, 15(1):253–289, April 1993.

[8] M. P. Jones. A system of constructor classes: Overloading and implicit higher-order polymorphism. In *Proc. of FPCA '93*, pages 52–61. ACM Press, 1993.

[9] M. P. Jones. Type classes with functional dependencies. In *Proc. of ESOP'00*, volume 1782 of *LNCS*. Springer-Verlag, 2000.

[10] S. Peyton Jones, M. P. Jones, and E. Meijer. Type classes: an exploration of the design space. In *Haskell Workshop*, June 1997.

[11] S. Kaes. Parametric overloading in polymorphic programming languages. In *In Proc. of ESOP'88*, volume 300 of *LNCS*, pages 131–141. Springer-Verlag, 1988.

[12] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM Press, 2004.

[13] R. Lämmel. A semantical approach to method-call interception. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 41–55. ACM Press, 2002.

[14] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical approach to generic programming. In *Proc. of ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003)*, pages 26–37. ACM Press, 2003.

[15] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Proc. of POPL '95*, pages 333–343. ACM Press, 1995.

[16] H. Masuhara, H. Tatsuzawa, and A. Yonezawa. Aspectual caml: an aspect-oriented functional language. In *Proc. of ICFP'05*, pages 320–330. ACM Press, 2005.

[17] S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[18] T. Sheard and S. Peyton Jones. Template meta-programming for Haskell. In *Proc. of the ACM SIGPLAN workshop on Haskell*, pages 1–16. ACM Press, 2002.

[19] P. J. Stuckey and M. Sulzmann. A theory of overloading. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(6):1–54, 2005.

[20] M. Sulzmann. AOP Haskell light: Aspect-oriented programming with type classes. http://www.comp.nus.edu.sg/~ sulzmann/aophaskell.

[21] S. R. Thatte. Semantics of type classes revisited. In *LFP '94: Proceedings of the 1994 ACM conference on LISP and functional programming*, pages 208–219. ACM Press, 1994.

[22] D. B. Tucker and S. Krishnamurthi. Pointcuts and advice in higher-order languages. In *Proc. of AOSD'03*, pages 158–167. ACM Press, 2003.

[23] P. Wadler and S. Blott. How to make *ad-hoc* polymorphism less *ad-hoc*. In *Proc. of POPL'89*, pages 60–76. ACM Press, 1989.

[24] D. Walker, S. Zdancewic, and J. Ligatti. A theory of aspects. In *Proc. of ICFP'03*, pages 127–139. ACM Press, 2003.

[25] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Trans. Program. Lang. Syst.*, 26(5):890–910, 2004.

[26] M. Wang, K. Chen, and S.C. Khoo. On the pursuit of staticness and coherence. In *FOAL '06: Foundations of Aspect-Oriented Languages*, 2006.

[27] M. Wang, K. Chen, and S.C. Khoo. Type-directed weaving of aspects for higher-order functional languages. In *Proc. of PEPM '06: Workshop on Partial Evaluation and Program Manipulation*, pages 78–87. ACM Press, 2006.

[28] G. Washburn and S. Weirich. Good advice for type-directed programming: Aspect-oriented programming and extensible generic functions. In *Proc. of the 2006 Workshop on Generic Programming (WGP'06)*, pages 33–44. ACM Press, 2006.