FOAL 2011 Proceedings

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

held at the
International Conference on
Aspect-Oriented Software Development

March 21, Porto de Galinhas, Pernambuco, Brazil

Gary T. Leavens, Shmuel Katz, and Hidehiko Masuhara (editors)

ACM International Conference Proceedings Series
ACM Press

ACM ISBN: 978-1-4503-0644-7/11/03

Contents

Preface

FOAL logos courtesy of Luca Cardelli

Aspect-oriented programming is a paradigm in software engineering and programming languages that promises better support for separation of concerns. The Tenth Foundations of Aspect-Oriented Languages (FOAL) workshop was held at the Tenth International Conference on Aspect-Oriented Software Development in Porto de Galinhas, Pernambuco, Brazil, on March 21, 2011. 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.

The workshop was organized by Gary T. Leavens (University of Central Florida, USA), Shmuel Katz (Technion–Israel Institute of Technology, Israel), and Hidehiko Masuhara (University of Tokyo, Japan). We are very grateful to the program committee, which was chaired very ably by Hridesh Rajan.

We thank the organizers of AOSD 2011 for hosting the workshop.

Message from the Program Committee Chair

The FOAL workshop in its tenth edition continues to attract foundational work on aspect-oriented software development. As in previous years, we were pleased to assemble yet another outstanding program committee for FOAL 2011.

The members of the program committee were:

- Hridesh Rajan — Program Committee Chair, Iowa State University, USA
- Werner M. Dietl — University of Washington
- Juergen Dingel — Queen's University
- Erik Ernst — University of Aarhus
- David Garlan — Carnegie Mellon University
- Atsushi Igarashi — Kyoto University
- Radha Jagadeesan — DePaul University
- Oscar Nierstrasz — University of Berne
- Bruno C. d. S. Oliveira — Seoul National University
- Jeremy Siek — University of Colorado, Boulder
- Neelam Soundarajan — Ohio State University
- Mario Südholt — École des Mines de Nantes
- Mitch Wand — Northeastern University

As in the past, each paper was subjected to full review by at least three reviewers between Jan. 21 and Jan. 31, 2011. Given the short turnaround time the FOAL 2011 program committee members worked really hard and provided authors with excellent and detailed reviews. I am grateful to the program committee members for their dedication, insightful comments, attention to detail, and the service they provided to the community and the individual authors.

I am also grateful to the authors of submitted works. Without these excellent submissions, a successful FOAL workshop may not be realized.

Finally, I would like to thank the other members of the organizing committee of FOAL — Gary T. Leavens, Shmuel Katz, and Hidehiko Masuhara — for their work in guiding us toward another inspiring workshop.

Hridesh Rajan
FOAL '11 Program Committee Chair
Iowa State University, USA

Applying Translucid Contracts for Modular Reasoning about Aspect and Object Oriented Events

Mehdi Bagherzadeh$^\beta$ Gary T. Leavens$^\theta$ Robert Dyer$^\beta$
$^\beta$Iowa State University
{mbagherz, rdyer}@iastate.edu
$^\theta$University of Central Florida
leavens@eecs.ucf.edu

ABSTRACT

The Implicit Invocation (II) architectural style improves modularity and is promoted by aspect-oriented (AO) languages and design patterns like Observer. However, it makes modular reasoning difficult, especially when reasoning about control effects of the advised code (subject). Our language Ptolemy, which was inspired by II languages, uses translucid contracts for modular reasoning about the control effects; however, this reasoning relies on Ptolemy's event model, which has explicit event announcement and declared event types. In this paper we investigate how to apply translucid contracts to reasoning about events in other AO languages and even non-AO languages like C#.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Programming by contract, Assertion checkers; F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Assertions, Invariant, Pre- and postconditions, Specification techniques

General Terms

Design, Languages, Verification

Keywords

Translucid contracts, modular reasoning, implicit invocation, aspect-oriented interfaces, grey-box specification, Ptolemy, quantified typed events, aspect-oriented events, object-oriented events

1. INTRODUCTION

Reasoning about the control effects of aspect-oriented (AO) programs seems difficult because: (1) join point shadows are pervasive, and (2) advice can have interesting control effects (e.g., throwing an exception or not proceeding) which are difficult to specify using black-box behavioral contracts. One way to avoid the first problem is to limit the application of advice to the base code. In our previous work on Ptolemy, join point shadows are limited to the places where events are explicitly announced [13]. To solve the second problem, we proposed translucid contracts [3]; these are grey-box based specifications limiting the behavior of advice. The grey-box nature of translucid contracts makes it possible to reveal some implementation details while hiding others.

In this paper we show the extent to which translucid contracts can be applied to several AO interface proposals as well as a non-AO language (C#). That is, we separate the ideas of translucid contracts from their original context, namely the Ptolemy language. The key features of Ptolemy that are relevant are explicitly declared event types, explicit event announcement and its quantification mechanism. Ptolemy's event announcement makes join point shadows in the base code, explicit. The quantification mechanism allows static computation of the set of advice at a specific place in the code.

Contributions of this work include:

- Application of translucid contracts to other AO interfaces, specifically crosscutting programming interfaces (XPI) [17], aspect-aware interfaces (AAI) [9] and Open Modules [1].
- A programming idiom to apply translucid contracts to a non-AO language with built-in support for events, C#.

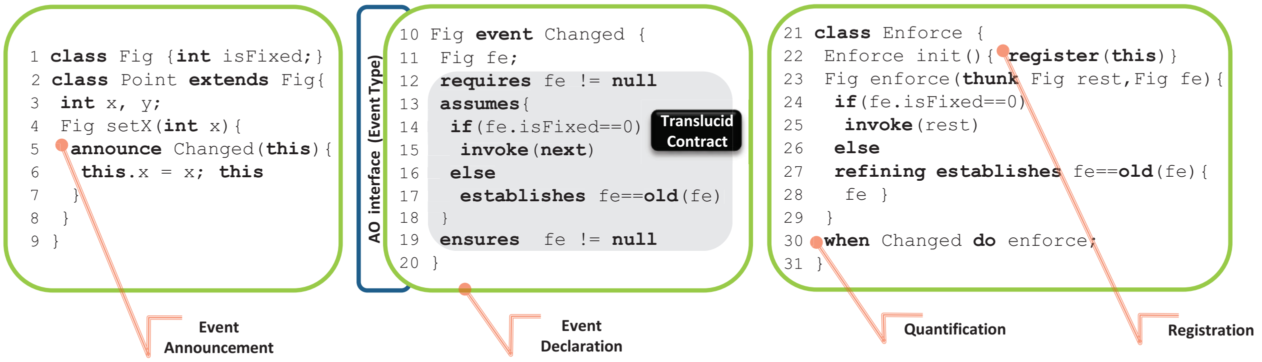
In the rest of the paper, Section 2 provides background information about translucid contracts in Ptolemy. Section 3 shows how to apply translucid contracts to other proposals for AO interfaces. Section 4 discusses a proposed programming idiom to apply translucid contracts to C# events. Section 5 discusses related work and finally Section 6 concludes the paper.

2. TRANSLUCID CONTRACTS IN PTOLEMY

The canonical figure editor example in Figure 1, illustrates translucid contracts in the Ptolemy language [13]. A figure element `Point` sets the value of its x-coordinate in method `setX`. The requirement in this example is: skip the modification of the x-coordinate, of the figure element point, if the figure element is *fixed* and not modifiable. This requirement could be implemented using event-driven programming techniques, which announce an event when `setX` is about to modify the `Point` and have an event handler method like `enforce` which enforces the non-modifiability requirement of the fixed figure element.

Our language Ptolemy, used in the implementation of the example in Figure 1, enables event-driven programming by the introduction of *quantified, typed events*. Event type `Changed` (lines 10-20) abstracts concrete events which represent modification to figure elements, such as points. Context variable `fe` (line 11) is a piece of information communicated between `Point` (subject), which announces `Changed`, and its handler `Enforce` (observer). The

FOAL'11, March 21, 2011, Pernambuco, Brazil.
Copyright 2011 ACM 978-1-4503-0644-7/11/03 ...$10.00.



```
1 class Fig {int isFixed;}
2 class Point extends Fig{
3  int x, y;
4  Fig setX(int x){
5   announce Changed(this){
6    this.x = x; this
7    }
8   }
9 }
```

```
10 Fig event Changed {
11  Fig fe;
12  requires fe != null
13  assumes{
14   if(fe.isFixed==0)
15    invoke(next)
16   else
17    establishes fe==old(fe)
18  }
19  ensures  fe != null
20 }
```

```
21 class Enforce {
22  Enforce init(){register(this)}
23  Fig enforce(thunk Fig rest,Fig fe){
24   if(fe.isFixed==0)
25    invoke(rest)
26   else
27   refining establishes fe==old(fe){
28    fe }
29  }
30 when Changed do enforce;
31 }
```

Figure 1: A translucid contract for the event type Changed

translucid contract (lines 12–19) limits the behavior of the refining handler methods like `enforce` using pre- and post-condition constraints phrased in **`requires`** and **`ensures`** clauses (lines 12 and 19). It also limits the control effects of the refining handlers by imposing structural constraints on their implementation using **`assumes`** block (lines 13–18). Subject `Point` announces event `Changed` explicitly using an **`announce`** expression (lines 5–7), passing the parameter **`this`** to be mapped to the context variable `fe`. Observer `Enforce` shows its interest in being notified about announcements of event `Changed` using the binding declaration **`when`** − **`do`** (line 30), which says to run method `enforce` whenever an event of type `Changed` is announced. The subject `Enforce` registers itself as an observer for event `Changed` using the **`register`** expression (line 22).

As mentioned earlier, translucid contracts restrict the control effects of the refining handlers by imposing constraints on the structure of the code in their implementation. Handlers of a specific event should refine the translucid contract of the event. The **`assumes`** block (lines 13–18) contains this information. Translucid contracts are more expressive compared to black-box contracts as they can reveal some implementation details about their refining handlers using *program expressions*, while hiding others using *specification expressions*. For example, the program expression (line 14) is conveying the fact that each refining handler must evaluate the **`if`** expression in its implementation as the very first expression followed by an **`invoke`** (line 15). While program expressions reveal implementation details, specification expressions (line 17) hide them, which allows for variability in the refining handlers' implementations. The programmer of the observer module, by just looking at the observer and the translucid contract, can conclude that if the figure element `fe` is not fixed then the handler method is called, allowing the modification of the figure (lines 14–15); otherwise the handler is skipped and the figure is not changed (line 17). **`invoke`** is Ptolemy's equivalent of AspectJ's **`proceed`**.

In terms of variability of the handlers, outside the scope of this example, structural constraints in the assumes block could be as liberal as **`establishes`** *true* which specifies any handler without an invoke expression in its body or **`establishes`** *true*; **`invoke`**(**`next`**); **`establishes`** *true* which allows any handler, with the invoke expression somewhere in its implementation.

Verification of the handler method's refinement of the translucid contracts is carried out via a hybrid static and dynamic approach. Static structural refinement checks for the textual matching between program expressions in the translucid contract and the handler implementation at the same structural positions in the code and the contract [3]. For example, lines 14–16 match lines 24–26. Specification expressions in the contract must be refined by **`refining`** expressions carrying the same specification. For example, line 17 is refined by the refining expression on lines 27-28. Runtime assertions assure that refining expressions actually refine the specification they claim to refine. Pre- and post-conditions of the translucid contract are also enforced using runtime probes inserted at the beginning and end of each handler and before and after event announcement.

The key point to notice when applying translucid contracts to the event types in Ptolemy, is that: *In Ptolemy, each handler knows about the type of events it handles, statically at compile time*. Thus, having the handler's implementation and the declaration of the event type it handles, refinement of the contract by the handler could be carried out modularly without any need for whole-program analysis. This is not the case in all languages with built-in event-driven mechanism such as C#. In these languages *handlers do not statically know about the type of events they might handle.* In this work, we propose a very simple programming idiom which allows the handlers to know about the type of events they handle, which in turn enables modular verification of their refinement of the translucid contract of the events they handle.

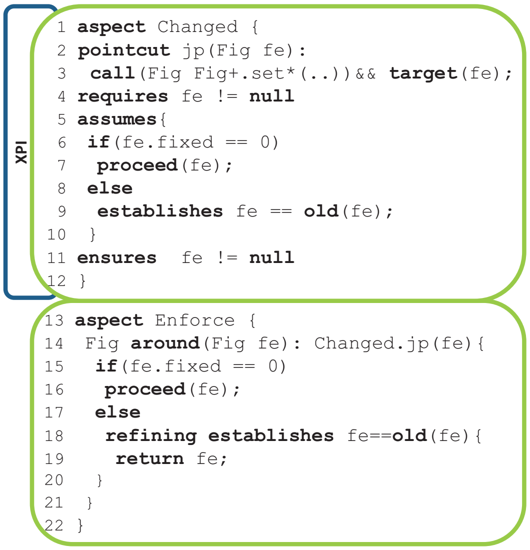
3. APPLICABILITY TO OTHER AO INTERFACES

As mentioned in Section 1, pervasive join point shadows are one of the obstacles in the modular reasoning about AO programs. AO interfaces tackle this problem by making join points explicit. Ptolemy's event types could be thought of as AO interfaces. We show the applicability of translucid contracts to crosscutting interfaces (XPI) [17], aspect-aware interfaces (AAI) [9], and Open Modules [1] and discuss changes in the refinement rules required to verify such programs. Other AO interfaces such as join point types (JPT) [16] and explicit join points (EJP) are not discussed as they are similar to Ptolemy's event types, discussed in our previous work [3]. For a more detailed discussion on the applicability of translucid contracts to AO interfaces see our previous work [2].

3.1 Translucid Contracts for XPIs

The key idea in crosscut programming interfaces (XPIs) [17] is to establish an interface, based on design rules, to decouple the base and the aspect design. An XPI limits the exposure of join points and also the behavior of advised and advising code using black-box contracts in terms of provides and requires clauses, with no mechanism to check the full compliance to the contract.

Figure 2 illustrates the applicability of translucid contracts to XPI `Changed` on lines 4–11, in an AspectJ implementation of the figure editor example introduced in Section 2. XPI `Changed`



```
 1 aspect Changed {
 2 pointcut jp(Fig fe):
 3  call(Fig Fig+.set*(..))&& target(fe);
 4 requires fe != null
 5 assumes{
 6  if(fe.fixed == 0)
 7   proceed(fe);
 8  else
 9   establishes fe == old(fe);
10  }
11 ensures  fe != null
12 }
13 aspect Enforce {
14  Fig around(Fig fe): Changed.jp(fe){
15   if(fe.fixed == 0)
16    proceed(fe);
17   else
18    refining establishes fe==old(fe){
19     return fe;
20    }
21   }
22 }
```

Figure 2: Applying translucid contract to XPI

and aspect Enforce in Figure 2 are the counterparts of Ptolemy's event type Changed and handler Enforce in Figure 1. The language for expressing translucid contracts is slightly adapted to use AspectJ's **proceed** instead of Ptolemy's **invoke**, on lines 7, 16.

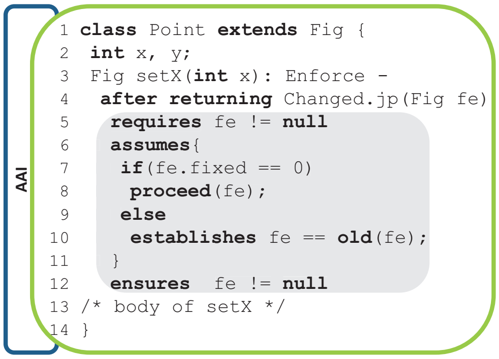
Unlike Ptolemy, where the translucid contract is attached to the event type (lines 12–19, Figure 1), in the XPI the contract is attached to the pointcut declaration (lines 4–11, Figure 2). In the Ptolemy example of Figure 1 only the context variable fe defined on line 11 could be accessed in the contracts, likewise in the XPI example, only the variable fe exposed by the pointcut (lines 2–3, Figure 2) is used in the contract. In Ptolemy the event type of interest is specified by the handler in the binding declaration (line 30, Figure 1) whereas in the XPI example, handler Enforce reuses the pointcut declaration in XPI Changed (line 14, Figure 2). Our refinement rules could be added here in the AO type system enforcing that the advice body on lines 15–21 must refine the translucid contract of the pointcut declaration on line 14. As it can be seen, the refinement rules are applicable to XPIs with only minor changes.

3.2 Translucid Contracts for AAIs

Some AO interfaces such as XPIs could be specified explicitly, whereas others such as aspect-aware interfaces (AAIs) [9] could be computed from the implementation, given whole-program information. Figure 3 illustrates the AAI for the figure editor example of Section 2. Figure 3 shows the extracted AAI for the method setX on lines 3–4 along with a translucid contract on lines 5–12, carried over from the pointcut to the join point shadow. In AAI the advised join point in method setX contain the details of the advising advice on lines 3–4. Syntax and refinement rules similar to XPIs are applicable here. Similar ideas can also be applied to aspect-oriented development tools such as AJDT, which provide AAI-like information at each join point shadow in an AspectJ program.

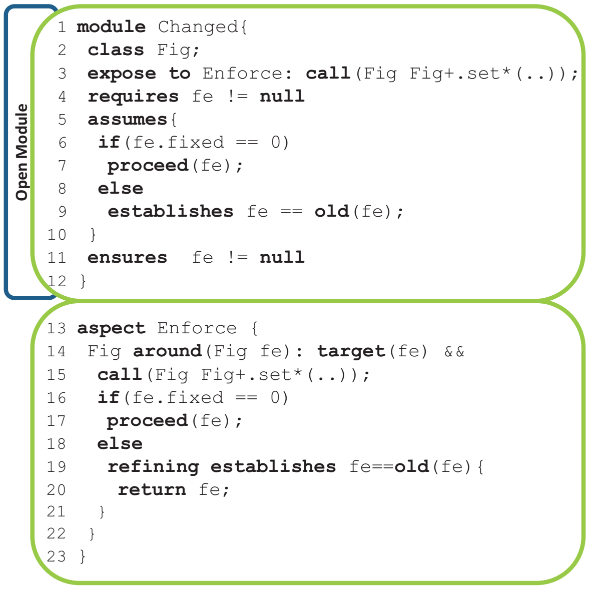
3.3 Translucid Contracts for Open Modules

Open Modules [1] allow explicit exposure of pointcuts for behavioral modifications by aspects, which is similar to signaling events using the announce expression in the Ptolemy. The implementations of these pointcuts remain hidden from the aspects which in turn reduces the impact of the base code changes on the aspect. However, in Open Modules, each explicitly declared pointcut has to be enumerated by the aspect for advising.



```
 1 class Point extends Fig {
 2  int x, y;
 3  Fig setX(int x): Enforce -
 4   after returning Changed.jp(Fig fe)
 5    requires fe != null
 6    assumes{
 7     if(fe.fixed == 0)
 8      proceed(fe);
 9     else
10      establishes fe == old(fe);
11    }
12    ensures  fe != null
13 /* body of setX */
14 }
```

Figure 3: Applying translucid contract to AAI



```
 1 module Changed{
 2  class Fig;
 3  expose to Enforce: call(Fig Fig+.set*(..));
 4  requires fe != null
 5  assumes{
 6   if(fe.fixed == 0)
 7    proceed(fe);
 8   else
 9    establishes fe == old(fe);
10  }
11  ensures  fe != null
12 }
13 aspect Enforce {
14  Fig around(Fig fe): target(fe) &&
15   call(Fig Fig+.set*(..));
16   if(fe.fixed == 0)
17    proceed(fe);
18   else
19    refining establishes fe==old(fe){
20     return fe;
21    }
22  }
23 }
```

Figure 4: Applying translucid contract to Open Modules

Figure 4 illustrates the applicability of translucid contracts, lines 4–11, to Open Module Changed in the figure editor example of Section 2. To retain similarity with other examples in the paper, syntax from Ongkingco *et al.*'s AspectJ implementation [12] is used in the example. Compare Open Module Changed and aspect Enforce with event type Changed and handler Enforce in Figure 1. Open Module Changed in Figure 4 exposes a pointcut of **class** Fig on line 2 which is only advisable by the aspect Enforce marked by **expose to**, line 3. The translucid contract on lines 4–11 limits the the interaction between Enforce and the pointcut exposed on line 3.

Like contracts in XPIs, in Open Modules the contract on lines 4–11 is attached to the pointcut declaration on line 3. Variable fe named in the contract is the one exposed by the pointcut on line 3, again like XPIs. The proposed rules for verifying refinement need to be modified slightly. In Ptolemy, the event type of interest Changed is specified in the binding declaration (line 30, Figure 1), whereas in the AspectJ implementation of Open Modules [12], aspects cannot reuse pointcuts exposed by the Open Module and need to enumerate the pointcut in the advice declaration again, lines 14–15. Refinement rules could be added here in the AO type system. The same adaptations in the syntax and refinement rules as of XPI's are applicable to Open Modules. The challenge is to match aspect Enforce pointcut definition on lines 14–15, with the Open Module one on line 3 to pull out its contract for refinement checking.

4. APPLICABILITY TO NON-AO LANGUAGES

Section 3 discussed the application of translucid contracts to AO interfaces rather than Ptolemy's event types. But the applicability of translucid contracts is not limited to just AO languages. In this section we discuss their applicability to a non-AO language, C#, with built-in support for event announcement and handling.

4.1 Problem

As discussed earlier in Section 1, Ptolemy's key feature for applicability of translucid contracts is that for any specific handler the set of potential events it handles is statically known. In other words, for each event type in Ptolemy, it is pretty straightforward to determine the set of its potential handlers using Ptolemy's quantification mechanism. Thus the translucid contract for the handler could be easily pulled out and refinement can be checked in a modular fashion using only the handler implementation and the contract.

In languages with built-in event announcement and handling, such as C#, the set of handlers for an event is not easily known statically. In C# the event model relies on type-safe method pointers (delegates) which could be used to dynamically register a method as a handler for a specific event. The signature of the handler often only includes the context variable and does not indicate the specific type of event being handled, such as:

```
Fig enforce (Fig fe);
```

This handler could handle multiple events, as long as the events pass in the context variable `fe` of type `Fig`. To determine the specific event being handled by each handler, we propose a simple programming idiom which *requires the event type to be passed as an argument to the handler method*. Using this idiom, by only looking at the handler method's signature, the type of event it handles can be easily determined. The idiom resembles the quantification mechanism in Ptolemy, as in line 30 in Figure 1.

4.2 Translucid Contracts for C#

In this section event declaration, announcement and handling in C# is illustrated and compared with Ptolemy using the figure editor example in Figure 1. The C# example is more verbose than needed in order to provide handlers with an **Invoke** statement which causes the next applicable handler to run, like its counterpart the invoke expression in Ptolemy. This section also discusses the proposed programming idiom. All our proposal requires is to pass into the handler the event type it handles, as a formal parameter.

```
10 class Changed:EventType <Fig, Changed.Context>{
11  class Context{
12   Fig fe;
13   Context (Fig fe){ this.fe = fe;}
14   Fig contract() {
15    Contract.Requires(fe != null);
16    Contract.Ensures(fe != null);
17    if (fe.isFixed==0)
18     return new Changed().Invoke();
19    else {
20     Contract.Assert(1==1);
21     Contract.Assert(fe==Contract.OldValue(fe));
22    }
23  }}}
```

Translucid Contract

Figure 5: Applying translucid contract to C#

Figure 5 illustrates declaration of event type `Changed`, similar to `Changed` in Figure 1, with return type `Fig`, line 10, and the context variable `fe`, defined on line 12 and set on line 13. Like Ptolemy, in C# the contracts are attached to the event type, lines 15–21. Method `contract` on lines 14–22 is the placeholder for the translucid contract. Lines 15–16 state pre- and post-conditions of the contract using the Embedded Contracts Language [6]. Lines 17–22 illustrate the body of the **assumes** block of Figure 1 lines 13–18. Lines 20–21 in Figure 5 are the equivalent of the specification expression of line 17 in Figure 1. Specification **establishes** fe == **old**(fe) is the sugar for **requires** $true$ **ensures** fe == **old**(fe). The **Invoke** method on line 18 causes the next applicable handler to run. It is provided by the class `EventType` in the C# library for Ptolemy, which is not shown here.

```
1 class Fig { int isFixed; }
2 class Point:Fig {
3  int x, y;
4  void setX(int x) {
5   Changed.Announce(new Changed.Context(this),()=>{
6    this.x = x;
7    return this;});
8  }
9 }
```

Figure 6: Event announcement with event types in C#

Figure 6 illustrates the subject `Point`. Compare it with class point in Figure 1. On line 5, `Point` announces the event `Changed` using the `Changed.Announce` method, similar to event announcement on line 5 of Figure 1. The receiver of the announce method is the event type being announced and the event body is provided as an anonymous lambda statement, lines 6–7. The context variable `fe` is created and set on line 5 by creating the object `Changed.Context`.

```
24 class Enforce {
25  Enforce init(Changed.Register(enforce);}
26  Fig enforce(EventType<Fig, Changed.Context next){
27  Contract.Requires(fe != null);
28  Contract.Ensures(fe != null);
29   if (next.fe.fixed == 0)
30    return next.Invoke();
31   else {
32    Contract.Assert(1==1);
33    return next.context().fe;
34    Contract.Assert(next.Context.fe ==
35         Contract.OldValue(next.Context.fe));
36   }}}
```

Figure 7: Event handler in C#

Figure 7 illustrates the handler method `enforce` on lines 26–36. Compare it with the `enforce` in Figure 1. Event registration is done via the call to the **register** method on the event type, line 25. The **Invoke** statement is similar to Ptolemy's invoke expression, allowing the next applicable handler to be called. Lines 32-35 are the equivalent of Ptolemy's refining expression on lines 27–28 of Figure 1. Assertion statements on lines 32 and 34–35 are run time probes added to enforce the specification stated by specification expression on lines 20–21 of Figure 5. Ptolemy's quantification mechanism is simulated in C# by the proposed idiom of passing the event type to the handler as a parameter, on line 26 .

4.3 Discussion

As previously mentioned in Section 2, runtime assertions assure that each handler method refines the pre- and post-condition of the event type it handles. They also check that Ptolemy's `refining` expression actually refines the specification it claims. In C# it means the insertion of runtime probes on lines 27–28 of Figure 7 to enforce the contract's pre- and post-conditions, stated on lines 15–16 of Figure 5. Also, the addition of assertions on lines 32 and 34-35 to make sure the specification expression on lines 20-21 of Figure 5 is not violated by any program expression which claims to refine it, line 33 of Figure 7. Insertion of runtime probes and structural refinement of the contract by handlers could be carried out by a simple source to source transformation. The transformation also makes sure that the refining handler methods and each code block constrained by a specification expression have one exit point to avoid unreachable code (line 33, Figure 7) . Structural similarity is crucial to structural refinement [3, 14].

5. RELATED WORK

This work, especially the internals of the translucid contracts, relates to works which propose: (1) behavioral contracts for aspects and (2) modular reasoning techniques for AO interfaces.

Behavioral contracts for Aspects:. Use of behavioral contracts to limit the behavior of aspects for the ease of reasoning is an accepted approach, exercised in the works such as crosscut programming interfaces (XPI) [8, 18], Pipa [19] and Cona [10, 15] among the others. XPI's informal contracts in terms of constraints for the advised and the advising code, Pipa's JML-like annotations and Cona's contracts for both aspects and objects are all behavioral contracts, which makes them incapable of specifying any control effect of interest. Furthermore, there is no verification mechanism proposed for XPI contracts.

Modular Reasoning for AO Interfaces:. Frequent join point shadows are one of the obstacles in modular reasoning about AO programs. Open Modules [1], explicit join points [7], join point types [16] and Ptolemy [13] tackle this problem by limiting the number of join point shadows as we have done in this work. However they do not provide any concrete specification and verification mechanism for reasoning.

Understanding the control effects of the advice is another problem in modular reasoning. "Harmless" advice [5] assumes aspects with no side effects. Categorizing the aspects as assistants (or spectators) [4], which can(not) enhance the behavior of the base code helps with reasoning. EffectiveAdvice [11] proposes explicit advice points and composition and its typed model enforces control and data flow properties. However, its non-AO core makes it difficult to adapt it to II, AO and Ptolemy as it lacks quantification.

6. CONCLUSION

Although implicit invocation (II) improves modularity, it makes modular reasoning difficult especially reasoning about control effects. In the previous work [3] translucid contracts were proposed to enable modular reasoning in Ptolemy. In this work, we show that translucid contracts are independent of their original context, Ptolemy, and are applicable to other AO interfaces. We also propose a simple programming idiom to enable application of translucid contracts to C#. The basic requirement when applying translucid contracts is: for each handler, it should be possible to statically tell which event types it handles. The proposed idiom meets this requirement. The idiom is simple and general and can be applied to other OO languages. Using the idiom makes it possible to know what events a handler method can handle. In summary, translucid contracts are independent of Ptolemy and are applicable to implicit AO and explicit OO event announcement models.

Acknowledgments

Bagherzadeh and Dyer were supported in part by NSF grant CCF-10-17334. The work of Leavens was supported in part by NSF grant CCF-10-17262.

7. REFERENCES

[1] J. Aldrich. Open modules: Modular reasoning about advice. In *ECOOP'05*.

[2] M. Bagherzadeh, H. Rajan, and G. T. Leavens. Translucid contracts for aspect-oriented interfaces. In *FOAL '10*.

[3] M. Bagherzadeh, H. Rajan, G. T. Leavens, and S. Mooney. Translucid contracts: Expressive specification and modular verification for aspect-oriented interfaces. In *AOSD '11*.

[4] C. Clifton, G. T. Leavens, and J. Noble. Ownership and effects for more effective reasoning about Aspects. In *ECOOP '07*.

[5] D. S. Dantas and D. Walker. Harmless advice. In *POPL'06*.

[6] M. Fähndrich, M. Barnett, and F. Logozzo. Embedded contract languages. SAC '10.

[7] K. J. Hoffman and P. Eugster. Bridging Java and AspectJ through explicit join points. In *PPPJ'07*.

[8] K. J. Sullivan *et al.* Information hiding interfaces for aspect-oriented design. In *ESEC/FSE'05*.

[9] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *ICSE'05*.

[10] D. H. Lorenz and T. Skotiniotis. Extending design by contract for aspect-oriented programming. *CoRR*, abs/cs/0501070, 2005.

[11] B. Oliveira, T. Schrijvers, and W. R. Cook. Effectiveadvice: Disciplined advice with explicit effects. In *AOSD'10*.

[12] N. Ongkingco, P. Avgustinov, J. Tibble, L. Hendren, O. de Moor, and G. Sittampalam. Adding Open Modules to AspectJ. In *AOSD'6*.

[13] H. Rajan and G. T. Leavens. Ptolemy: A language with quantified, typed events. In *ECOOP'08*.

[14] S. M. Shaner, G. T. Leavens, and D. A. Naumann. Modular verification of higher-order methods with mandatory calls specified by model programs. In *OOPSLA'07*.

[15] T. Skotiniotis and D. H. Lorenz. Cona: Aspects for contracts and contracts for aspects. In *OOPSLA'04*.

[16] F. Steimann, T. Pawlitzki, S. Apel, and C. Kastner. Types and modularity for implicit invocation with implicit announcement. *TOSEM*, 20(1), 2010.

[17] K. J. Sullivan, W. G. Griswold, H. Rajan, Y. Song, Y. Cai, M. Shonle, and N. Tewari. Modular aspect-oriented design with XPIs. *TOSEM*, 20(2), 2009.

[18] W. G. Griswold *et al.* Modular software design with crosscutting interfaces. *IEEE Software'06*.

[19] J. Zhao and M. Rinard. Pipa: A behavioral interface specification language for AspectJ. In *FASE'03*.

Compositional Verification of Events and Observers (Summary)

Cynthia Disenfeld and Shmuel Katz
Department of Computer Science
Technion - Israel Institute of Technology
{cdisenfe,katz}@cs.technion.ac.il

ABSTRACT

By distinguishing between events and aspects, it is possible to separate the problem of identifying when an aspect should be applied, from what it must do. Observers (aspects that do not affect the state of the base system) are already part of aspect-oriented programming and language support is emerging for events that gather information and announce occurrence. The goal of compositional verification of events and observers is to prove that they are correct so that their guarantees may be used by other events or aspects. Moreover, a compositional verification model allows applying formal verification techniques in smaller models, and also building a library of events, in which for any base system that satisfies certain assumptions, the event detection will satisfy its guarantees. In this work compositional verification of events and observers will be defined to aid in the design of a framework that allows users to verify events, providing as well flexibility in the input language of the specification.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.4 [**Software Engineering**]: Software/Program Verification—*Correctness proofs, Model checking*

General Terms

Languages, Verification

Keywords

Events, Observer Aspects, Verification, Composition

1. INTRODUCTION

The goals of this work are to (1) precisely identify the components involved in the verification of events, (2) provide a methodological way to specify and verify events compositionally and (3) outline a framework design in which the input to the verification process that the user must provide is clearly defined, as are the steps performed automatically to verify events. The full version of this summary is available from the authors.

Aspect oriented programming (AOP) [13] allows expressing crosscutting concerns to the application in a modular way. AspectJ [12] defines a set of possible *joinpoints* - states where an aspect should be applied. For each aspect, *pointcuts* define where the response should be applied, and *advices* define what must be done.

However, AspectJ does not provide an optimal notation for a variety of problems. Most pointcuts in AspectJ can only see the present state in the execution and the current call stack. This does not give enough flexibility to be able to aggregate the history of events that have occurred. The second problem is the difficulty to share information between events: pointcuts only expose information on the target class, the arguments and the current aspect being executed. The third problem is that pointcuts are defined by means of events in the code, and sometimes we may be interested in expressing matching joinpoints in a more abstract way, for instance by defining events that occur as a result of the composition of other events.

[1, 17, 4] deal with the first problem by using a restriction of the language of aspects to regular expressions, or treating sequences of events but still the composition of lower level events and independence between the joinpoint and the response are not treated.

Douence *et al.* [6] present a solution for these problems by allowing to share variables between *crosscuts* (pointcuts), preserving the history of execution and defining composition between aspects. However the crosscuts are still tightly related to the *inserts* (advices), and this restricts reusability.

The separation of events has been presented already in the event-based approaches of [8, 14], independently of aspects.

[15, 11, 5] have identified the need of defining event aspects, spectators or observers that gather information but do not change the base system, although those aspects are allowed to print values.

Bockisch *et al.* [3] introduce a solution to these problems by syntactically distinguishing between *events* and aspects. Event declarations may accumulate information and do not affect the underlying system in any way, including printing values. They indicate when a certain concern should be woven and provide collected information of the system to be used by other event declarations or aspects.

Thus, the idea of defining aspects or events that collect information and are triggered when the collected informa-

FOAL'11, March 21, 2011, Pernambuco, Brazil.

tion satisfies a certain property is natural in systems. We will focus here on showing how to verify the correctness of event declarations, that use other events in their declaration. These definitions are also useful for existing systems already defined using observers and spectators even though the notation presented for events in [3] will be used here.

Events and observer aspects may seem trivial due to their spectative behavior on the base system. However, events incorporate the logic of when they must be triggered, and what information is exposed. They (and observer aspects) collect information from different possible sequences in the base system. This information may be collected from actions on the base system, and even subjected to some internal processing. Given that in the extension presented in [3] general aspects now respond to states in which events are detected, for later aspect verification it will be essential to assume that events are detected in the correct states and that they expose the expected information.

In this work we will focus on model checking techniques. However, present methods are "flat" in that they relate to aspects that directly are woven to a base system without event dependencies defined hierarchically. Here we emphasize the incorporation of assumed properties on used events in the verification of an event. Moreover, standard use of temporal logic assertions is problematic for event specifications. Thus, the framework presented gives enough liberty to choose among different modeling languages for the specification: regular expressions, Kripke models, Moore machines or temporal logic formulas.

The verification of events will be presented using the assume-guarantee model, which allows building a reference library of available events already verified. Hence, for any base system and set of events which satisfy the assumptions of needed events from the library, the library events may be included and their guarantees will hold without further verification.

1.1 Outline

This work is organized as follows: Section 2 presents background on models and simulations. Section 3 presents the definition of events and the necessary assumptions. In section 4 the verification steps are defined. Lastly, in Section 5 the conclusions are presented.

2. BACKGROUND

To be able to give a formal model of the specification and apply formal verification, the definitions of structures, homomorphism, preorder in structures, and Moore Machines presented in [10] will be used. The concepts of LTL, fairness and reductions presented in [7] will be used as well.

2.1 Kripke model for a Moore machine

In [10], a procedure was presented to obtain the corresponding Kripke structure of a given Moore machine. This structure contains the Cartesian product of each state with all the transition labels that may lead to another state. In our case we use a different construction, in which each state contains the transition labels that caused arriving to it. The formal definition is presented in the full version of this summary.

2.2 Model restriction

For a model $M = \langle S, S_0, R, L, \mathcal{F} \rangle$ given by a Kripke model over the set of atomic propositions AP, the operator $M \upharpoonright AP'$ represents the restriction of the model to the atomic propositions in AP'. That is, for all $s \in S$, $L'(s) = L(s) \cap AP'$.

3. EVENTS

Events collect information on the base system, are triggered when an interesting situation to be detected occurs, and may expose certain information.

3.1 Assumptions

The following is assumed:

- Event invocation and execution do not affect any variable external to the event declaration, and they also do not affect the control flow (they must return the execution flow to the base system at the point they were begun).
- Event internal fields are only updated within the event declaration execution.
- There are no cycles in the event dependencies, i.e., an event cannot depend on itself being triggered in the correct places or its own exposed information being correct.
- Fairness restrictions must satisfy that it is always possible to get to a returning state for each event evaluation.

The first two properties may be checked by applying static analysis tools, adapting the tools presented in [2, 5, 16, 18] which work for identifying spectative or observer aspects. The dependency between events define an Event Dependency Graph [3], and cycle dependencies can be checked by analyzing this graph. Event models should guarantee that the only fair paths are those that eventually reach the end of execution.

From now on, the previously mentioned properties are assumed to hold.

3.2 Event Model

Each event contains a set of internal atomic propositions corresponding to the values of the internal fields, a set of external atomic propositions representing the parameters obtained from the lower level events, the initial values for the internal atomic propositions, and which *basic units* form the event.

Each *basic unit* u is a pair of a condition (consisting of other events having been triggered, pointcuts or predicates over the atomic propositions) and an event response that may only change the internal atomic propositions, or may trigger the event.

For the variables, fields, and parameters exposed, standard encoding and abstracting mechanisms are used, for example, range of values, boolean variables, etc.

We will use a high-level-syntax event example. There are well known translations from this language to the model form. The fragment of code presented in Figure 1 serves as an example of an event defined in terms of another event. There are three event declarations: (1) commit, (2) $TwoCommits$ is an event detected every two times *commit* is applied, and (3) $SixCommits$ is an event detected every six times *commit* is applied (but is defined indirectly using $TwoCommits$).

```
event commit(): call(* *.commit(..));
event TwoCommits()
    int counter=0;
    when(): commit()
        counter++;
        if (counter mod 2 == 0)
            trigger();
            counter = 0;

event SixCommits()
    int counter=0;
    when(): TwoCommits()
        counter++;
        if (counter mod 3 == 0)
            trigger();
            counter=0;
```

Figure 1: *SixCommits*

4. VERIFICATION PROCESS

4.1 Event evaluation Semantics

To be able to introduce the event evaluation semantics, the operator $\circledast$ is formally defined in the full version of this article. This operator represents the evaluation of the events in a set of events E in a base system or base system assumption B.

In this semantics, all the events are evaluated immediately, adding which events are detected to the atomic propositions of each possible state in the base system.

The base system regards events as being evaluated all at once, and in parallel due to their spectative nature. This differs from the weaving of aspects into a base system as presented in [9], where every state in the execution of the aspect is added to the woven model. Aspects are not instantly evaluated as their execution is not necessarily spectative and the state of the base system may change.

Note that in Figure 1 the basic units in the events take several steps to execute. However, in the resulting semantics event execution may be seen as immediate relative to the base system because of the locality of the fields, that no event affects the internal propositions of other events, and that there are no dependencies cycles.

Events and observers detect interesting situations in the base system, or collect information for certain paths of execution without affecting the state of the base system. This is one of the more useful advantages in restricting verification to events and observers rather than general aspects. Even though the size of the model grows due to the possible internal states of the events, the execution of all the events involved does not have to be represented at once, but only the resulting internal states and detected events.

From now on, $B \circledast E$ represents the base system with the detection of all events in E. U will also serve to denote a set of event declarations.

4.2 Specification

4.2.1 General idea

To prove a guarantee about an event E, E needs to assume the correctness of the guarantees of events it uses (and the same is true for a general aspect A).

The properties that an event must be proved to satisfy may be categorized as:

1. The event is triggered in the correct places. This requires defining exactly which sequence of situations and contexts in the base system and previously verified events should cause the current event to be triggered. The specification is in terms of event detections, exposed parameters and may as well include auxiliary variables.
2. The parameters exposed by the event satisfy the intended relations with the history of execution.

4.2.2 Specification definition

An event's specification, $\langle Ass, Guar\rangle$ - representing the assume and the guarantee - may be given in different specification languages. If the specification is given as a regular expression, then the equivalent automaton is obtained and it can be understood as a Moore Machine. If any of them is given as a Moore machine, then the equivalent Kripke model is built. If any of them is given as a CTL or LTL formula, then its tableau is built.

In particular the specification of $TwoCommits$ may be expressed as $\langle Ass, Guar\rangle$ where $Ass \equiv \neg commit$ (*commit* is false in the initial state) and *Guar* is given in Figure 2. This guarantee represents that every two occurrences of *commit*, *twocommits* hold.

The guarantee of the event $TwoCommits$ should be the assumption of $SixCommits$. Then, $SixCommits$' specification is given by $\langle Guar_{TwoCommits}, Guar_{SixCommits}\rangle$.

The guarantee of $SixCommits$ can be expressed as a Moore machine, as in Figure 4 or as its equivalent Kripke model presented in Figure 5.

When the event is intended to occur dependent on the occurrence of other events, either after a sequence of other events or the lack of events, the preferred specification is as a regular expression of events or in state machines, where both the specification and the event are given in that form. Aspect specifications usually refer to what properties each state must satisfy. Events, on the other hand, do not modify the state of the base system and are specified by means of what sequences of triggered event lead to them being detected, and what properties their exposed parameters must satisfy. Hence, aspects usually satisfy properties given in temporal logic, over the atomic propositions that represent the state, and events preserve the state so they refer to sequences of states instead. For every possible sequence it must be described whether it leads (or not) to the event to be detected and which information is to be exposed. Temporal logic expressions can be used to specify events, but become awkward and unreadable very quickly when sequences of lower level events must be expressed. Therefore we prefer regular expressions or state machines.

Specifying a property that the parameters satisfy when the event is detected may be defined by means of any of the languages presented for specification and a similar verification process can be applied.

4.3 Verification

Event verification consists in checking whether: given the assumption *Ass* of E on the base system and used event detectors, when detecting the event E, the guarantee *Guar* is satisfied (expressing both when E is detected and what must hold for parameters it then exposes).

In more formal notation, the goal in event verification is to prove that $B \circledast E \models Guar$. This is, the base system together with the event detection satisfies its specification.

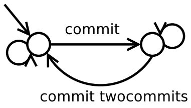


Figure 2: Moore machine: $Guar_{TwoCommits}$

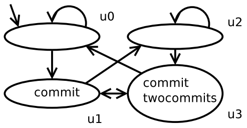


Figure 3: Kripke model: $Guar_{TwoCommits}$

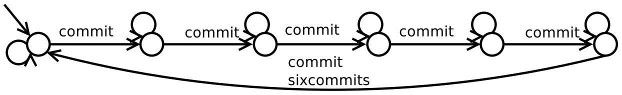


Figure 4: Moore machine: $Guar_{SixCommits}$

However, in order to obtain a modular verification of events, the assume guarantee model is used and the specification will be given by $\langle Ass, Guar\rangle$. The goal will be to prove that if Ass is the assumption about the base system and used event detectors, and $Ass \circledast E \models Guar$, then for any base system B and set of used event declarations U such that $B \circledast U \models Ass$, it can be inferred that $B \circledast U \circledast E \models Guar$, this is, the base system with all the event detectors incorporated satisfies its guarantee. At this step, verification of events is presented for a base system which has no aspects that may affect the event woven into it. Verification of events together with aspects woven will be analyzed in future work.

In particular, for the correctness in the detection of the event, if there exists an assumption Ass_B on the base system such that:

$$\begin{aligned} B &\preceq Ass_B \\ Ass_B \circledast U \upharpoonright AP_{Ass} &\equiv Ass && (1) \\ Ass \circledast E \upharpoonright AP_{Guar} &\equiv Guar && (2) \end{aligned}$$

Then:

$$\begin{aligned} Ass_B \circledast U \circledast E \upharpoonright AP_{Guar} &\equiv Guar \\ B \circledast U \circledast E &\preceq Guar \end{aligned}$$

The previous inference expresses that:

- Given a base system that satisfies a certain assumption Ass_B
- Given that the composition of this assumption with the set of used event detectors is equivalent to the assumption of the event, and

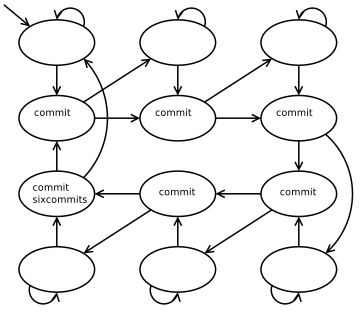


Figure 5: Kripke model: $Guar_{SixCommits}$

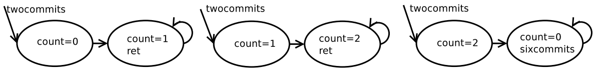


Figure 6: $SixCommits$

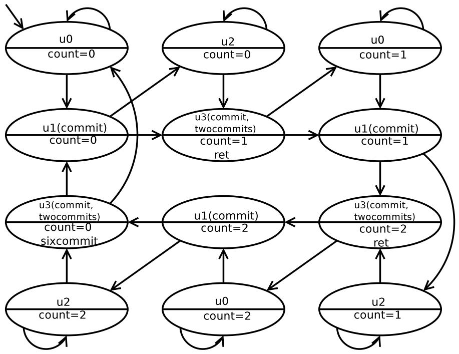


Figure 7: $Guar_{TwoCommits} \circledast SixCommits$

- Given that an event E is proven to be correct with respect to its specification

Then, the composition of the base system, used event detectors and E satisfies the guarantee $Guar$.

In the previous inference, (1) may be proven using the assume-guarantee model as well. In (1) and (2), proving the bisimulation guarantees that there exist paths in the model on the left side of the equation that behave as presented in the guarantee. Ass, $Guar$ and E must be given by the user so as to verify that E satisfies its specification $\langle Ass, Guar\rangle$.

Considering the example, a base system B will be composed with $TwoCommits$ and $SixCommits$. The first step is to prove $TwoCommits$ is correct. When $\neg commit$ is initially true, the tableau of the assumption together with the model of the code of $TwoCommits$ can be proven to satisfy Figure 3. Turning to $SixCommits$, when its assumption holds, the model of the code of $SixCommits$ can be proven to satisfy its guarantee as presented below.

The event $SixCommits$ - with $AP_0 = \{count = 0\}$ contains only one basic unit modeled in Figure 6. The event is evaluated for each occurrence of $twocommits$. Every three occurrences of $twocommits$, $sixcommits$ is triggered.

The event evaluation of $SixCommits$ is given by:

$S = \{u_0, u_1, u_2, u_3\} \times 2^{AP_{SC}}$ where $\{u_0, u_1, u_2, u_3\}$ are the states from Figure 3 and $AP_{SC} = \{count = 0, count = 1, count = 2, ret, sixcommits\}$. $S_0 = \{(u_0, count = 0)\}$

Considering the definition of the relation between the states and the event, the model in Figure 7 is obtained.

Restricting the model of $Guar_{TwoCommits} \circledast SixCommits$ to the atomic propositions of the specification, it is exactly the same model as given by the specification.

Therefore, effectively there is a homomorphism such that $Guar_{TwoCommits} \circledast SixCommits \equiv Guar_{SixCommits}$.

The previous procedure shows the correctness of $SixCommits$ with respect to its specification. Now, for any base system B that satisfies $\neg commit$, $B \circledast TwoCommits$ is correct. Moreover, for any implementation of $TwoCommits$ that satisfies the guarantee presented in Figure 3, incorporating the evaluation of $SixCommits$ will satisfy $Guar_{SixCommits}$. Consequently, it can be inferred that for any base system B

such that *commit* does not occur in the initial state, and for the models of the code presented for $TwoCommits$ and $SixCommits$:

$$B \circledast TwoCommits \circledast SixCommits \vDash Guar_{SixCommits}$$

Note how the guarantee of a simpler event is used as an assumption of one that uses it, and is incorporated into the verification.

The previous verification may include information on what values are exposed by the parameters. For the correctness of the exposed parameters a simulation is enough to prove the correctness and conclude that $B \circledast U \circledast E \preceq Guar$.

The process of finding a homomorphism for $\preceq$ can be done automatically by the algorithm presented in [10]. There are also automatic methods for obtaining the structure equivalent to a Moore machine or the tableau of a formula. The only things remaining to the user to provide are the specification and the model of the event expected to be verified.

5. CONCLUSIONS

Given the need to separate and abstract *when* an aspect is applied from *what* aspects do, *events* were incorporated in [3] aiming to identify *when* things should happen, and being able to collect information, or be detected when particular sequences of other events occurrences. Moreover, observers and spectator aspects [5, 11] are part of current programming practices in aspect-oriented programming. Due to their spectative nature, events and observers may seem trivial to be verified for correctness. However, other events and aspects may use the information and detection of the event, hence events must be verified to be correct when they are triggered, and the information exposed must satisfy requirements that other entities depend on.

Events may be thought of as spectative aspects with the additional *triggering* action, the use of other events as conditions, and restricted not to have output. This guarantees non-interference between the events. No event can affect another event - except by triggering itself, and none of them may modify the state in the base system.

In this work a modular verification method for events is introduced where the user is requested to present the assumptions, the expected guarantees and the event itself to be verified. Without additional intervention of the user the property is verified relative to the specifications of the used events. The use of the guarantees of events as assumptions for other events is shown as well. For all the steps presented there are existing tools that perform the necessary algorithms.

The users may follow the procedures presented in the article to define their own specifications or use their own methods, using for example regular expressions or state machines. Expressing sequences of events in a temporal logic formula gets hard to read very easily.

In the full version of this summary (as noted, available from the authors), a fuller semantic notation is presented, as well as an additional example with $Guar$ on the parameters, and detection involving the absence of other events. In future work, the influence of events on aspects will be analyzed, to give a complete hierarchical and compositional formal verification algorithm for systems that include both events and aspects. In this framework, as opposed to [9], the assumptions about other events can be incorporated naturally.

6. REFERENCES

[1] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to aspectj. *SIGPLAN Not.*, 40:345–364, 2005.

[2] Y. Alperin-Tsimerman and S. Katz. Dataflow analysis for properties of aspect systems. In *Proceedings of 5th Haifa Verification Conference, LNCS 6405*, 2009.

[3] C. Bockisch, S. Malakuti, M. Aksit, and S. Katz. Making aspects natural - events and composition. In *AOSD 2011 Modularity Visions Track*, 2011.

[4] E. Bodden and V. Stolz. Tracechecks: Defining semantic interfaces with temporal logic. In *Software Composition*, 2006.

[5] C. Clifton and G. T. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. Technical report, Iowa State University, Department of Computer Science, 2002.

[6] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In *AOSD 2004*, 2004.

[7] J. Edmund M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[8] O. Etzion and P. Niblett. *Event Processing in Action*. Manning Press, 2010.

[9] M. Goldman, E. Katz, and S. Katz. Maven: modular aspect verification and interference analysis. *Formal Methods in System Design*, 37:61–92, 2010.

[10] O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Trans. Program. Lang. Syst.*, 16:843–871, 1994.

[11] S. Katz. Aspect categories and classes of temporal properties. *Transactions on Aspect-Oriented Software Development I*, LNCS 3880:106–134, 2006.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *ECOOP*, 2001.

[13] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*, 1997.

[14] D. C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., 2001.

[15] O. Mishali and S. Katz. The highspectj framework. In *Proc. of the 8th workshop on Aspects, Components, and Patterns for Infrastructure Software*, 2009.

[16] M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In *Proc. of the 12th ACM SIGSOFT Symp. on Foundations of Software Engineering*, 2004.

[17] W. Vanderperren, D. Suvée, M. A. Cibrán, and B. D. Fraine. Stateful aspects in jasco. In *Software Composition*, 2005.

[18] N. Weston, F. Taiani, and A. Rashid. Interaction analysis for fault-tolerance in aspect-oriented programming. MeMoT'07, 2007.

Supporting covariant return types and generics in type relaxed weaving

Tomoyuki Aotani
Japan Advanced Institute of Science and Technology
aotani@jaist.ac.jp

Manabu Toyama
University of Tokyo
touyama@graco.c.u-tokyo.ac.jp

Hidehiko Masuhara
University of Tokyo
masuhara@acm.org

ABSTRACT

This paper introduces our ongoing study on type safety of the type relaxed weaving mechanism in the presence of two Java 5 features, namely covariant return types and generics. We point out additional conditions that are necessary to ensure type safety, which can be checked by a slightly modified type checking rules for the type relaxed weaving.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-Oriented Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages

Keywords

Aspect-oriented programming, Type relaxed weaving, Covariant return types

1. INTRODUCTION

The around advice is one of the unique and important features in the aspect-oriented programming (AOP) languages based on the pointcut and advice mechanism [9] such as AspectJ [4,7]. It allows us to change the receiver and argument values of method/constructor calls, and also to replace operations with others without modifying the source code of the program. There has been several studies that address improving generality and/or applicability of around advice [3, 8], as well as those design a formal calculus and study type safety for AOP languages with around advice [1, 2, 6].

The type relaxed weaving [8] is a bytecode-level weaving mechanism for AspectJ family of languages. It improves applicability of around advice. It allows a piece of around advice to have a different return type from those of the join points where it is woven. We call such advice *type-relaxing advice* in this paper. Type safety of the type relaxed weaving is proved formally based on an object-oriented calculus called Featherweight Java for Relaxation (FJR) [8], which is an extension to Featherweight Java [5].

This paper introduces our ongoing study on type safety of the type relaxed weaving in the presence of advanced language features that FJR does not have, especially, covariant return types and generics. The covariant return type extension allows a class to override a method with a return type smaller (more specific) than that of the method in its superclass. The generics feature enables us to define generic classes and methods through type parametrization.

The contributions of the paper are as follows:

- We point out additional conditions that are necessary to ensure type safety of the type relaxed weaving in the presence of covariant return types and generics.
- We show that a small modification to the constraint generation algorithm for the original type relaxed weaving is sufficient to support covariant return types.

The rest of the paper is organized as follows. We first visit the type relaxed weaving and see the conditions that advice should satisfy in Section 2. Section 3 shows the problem to support covariant return types and generics. Section 4 explains the basic idea of our solution along with our type-checking algorithm. Section 6 concludes the paper.

2. TYPE RELAXED WEAVING

The type relaxed weaving [8] is a type-safe bytecode-level weaving mechanism for AspectJ. It allows around advice to have a different return type from the join point shadows' on which it is woven. More specifically, it relaxes the typing rule in AspectJ that restricts the return type of a piece of around advice to either the return type of its target join points or one of its subtypes.

Figure 1 is a simple example allowed by the type relaxed weaving, but not allowed by AspectJ. Lines 1–9 are skeletons of three Java classes `Object`, `Integer` and `BigInteger`. Lines 11–18 define the interface `Stream` and two classes `IntegerStream` and `BigIntegerStream`. Lines 20–26 defines a method that creates a `BigIntegerStream` object, picks up a `BigInteger` object from it and converts it into a `String` object. The boxed string at line 23 is the signature of the method call. Finally lines 29–31 define a piece of around advice that replaces a `BigIntegerStream` object with an `IntegerStream` object upon creation.

If we ignore the static types of local variables, it is safe to replace the expression `new BigIntegerStream()` at line

FOAL'11, March 21, 2011, Pernambuco, Brazil.

```
 1 // Skeletons of Java classes
 2 class Object{ String toString(){...} }
 3 class Integer extends Object{
 4   String toString(){...}
 5 }
 6 class BigInteger extends Object{
 7   String toString(){...}
 8   BigInteger abs(){...}
 9 }

11 // Definitions of stream classes
12 interface Stream{ Object get(); }
13 class IntegerStream implements Stream{
14   Object get(){...}
15 }
16 class BigIntegerStream implements Stream{
17   Object get(){...}
18 }

20 //in a class
21 void m(){
22   BigIntegerStream bs = new BigIntegerStream();
23   Object o = bs.get(); [Object BigIntegerStream.get()]
24   String s = o.toString();
25   /* s is never used below */
26 }

28 //in an aspect
29 IntegerStream around():call(BigIntegerStream.new()){
30   return new IntegerStream();
31 }
```

Figure 1: Streams and type-relaxing advice

22 with `new IntegerStream()`. This is because each of the classes is a subtype of `Stream` and the resulting object is used only as a `Stream` object. Note that the assumption is reasonable since local variables have no type information at bytecode-level in Java.

Intuitively, the type relaxed weaving checks such conditions in a Java bytecode program. Given a piece of around advice `a` and a join point `jp` where `a` is applied to, it checks consistency between the return type of `a` and the operations that use the return value from `jp`. The usages are a method (or constructor) call parameter, a method call target, a return value from a method, a field access target, an assigned value to a field, an array access target and an exception to throw.

In the example, the return type of the advice is `IntegerStream`. The join point is `new BigIntegerStream()` at line 22. The return value from the join point is used as the target of a method call (line 23) whose signature is `Object BigIntegerStream.get()`.

We can safely change the receiver's type in the signature to `Stream` because the former overrides the latter and it does not change the behavior of the program with respect to the semantics of `invokevirtual`/`invokeinterface`. For the same reason, we can safely invoke `Stream.get()` on an `IntegerStream` object.

3. PROBLEM

```
 1 // Redefining IntegerStream and BigIntegerStream
 2 class IntegerStream{
 3   Integer get(){...} //refining return type
 4 }
 5 class BigIntegerStream{
 6   BigIntegerStream get(){...} //refining return type
 7 }
 8 //in a class
 9 void m(){
10   BigIntegerStream s = new BigIntegerStream();
11   BigInteger i = s.get(); [BigInteger BigIntegerStream.get()]
12   BigInteger absi = i.abs();
13   /* s is never used */
14 }
15 /* and the same aspect*/
```

Figure 2: An example using covariant return types that goes type-unsafe after the advice in Figure 1 is woven

The type relaxed weaving is based on the Java 1.4 language, which lacks recent features covariant return types and generics. Supporting those features in type relaxed weaving is not straightforward as we discuss below.

3.1 Support for covariant return types

It is not enough to care about the usage of the return value from a join point when a class can override its superclass's method with a smaller return type. In this section we assume that the language that employs the type relaxed weaving, i.e., RelaxAJ [8], is slightly extended so that it can accept covariance of the return type of a method in the base code.

Figure 2 is a part of the program modified from Figure 1 in which `IntegerStream` and `BigIntegerStream` override the method `get` with the smaller return types, namely `Integer` and `BigInteger` in `IntegerStream` and `BigIntegerStream`, respectively. The method `m` is also changed so that it calls `abs` defined in `BigInteger`.

The around advice shown in Figure 1 can still be woven on the join point shadow `new BigIntegerStream()` at line 10 in Figure 2 because the return value from the join point is only used as the receiver object to invoke `BigIntegerStream.get()`, which overrides `Stream.get()`. Again invoking `Stream.get()` on `IntegerStream` is safe and thus the condition of the type relaxed weaving is satisfied.

The woven code is, however, no longer type-safe. In fact, invoking `abs` at line 12 fails because the receiver `s` is now an `Integer` object, which is the return value of `get` invoked on the return value of the around advice, that is, `IntegerStream`.

3.2 Support for generics

Relaxing return types with type parameters has the same problem to the covariant return types case.

Figure 3 shows an example from which the type relaxed weaving would generate type-unsafe code by applying type-relaxing advice. Lines 3–7 define a generic `Stream` class, which is intended to be used instead of the `Stream` class and the classes implementing it defined in the previous exam-

```
 1 ...
 2 // Stream class using generics
 3 class Stream<X>{
 4   X x;
 5   Stream(X x){ this.x=x; }
 6   X get(){...}
 7 }
 8 ...
 9 //in some class
10 void m(){
11   Stream<BigInteger> s =
12     new Stream<BigInteger>(new BigInteger("0"));
13   BigInteger i = s.get();             Object Stream.get()
14   BigInteger absi = i.abs();  BigInteger BigInteger.abs()
15   /* s is never used below */
16 }

18 //in some aspect
19 Stream<Integer> around():
20   call(Stream.new(Object))&&!within(/*the aspect*/){
21   return new Stream<Integer>(new Integer("0"));
22 }
```

Figure 3: An example using generics that goes type-unsafe after the advice is woven

ples. The method `m` is also modified so that it now uses the generic `Stream` class instead of `BigIntegerStream`. It invokes the method `get` at line 13, whose signature is `Object Stream.get()`.

Here we cannot know with which type the type parameter `X` of `Stream<X>` should be replaced because such information is erased in Java bytecode. The around advice is modified similarly (lines 19–22), i.e., its pointcut specifies creations of `Stream` objects and it returns a `Stream<Int>` object.

Because the pointcut matches `new Stream` at line 12 and `Object Stream.get()` can be invoked on a `Stream<Int>` object, it is allowed to weave the advice on the shadow.

The generated code is again no longer type-safe; it has an invocation to `abs` on `Integer`, which always fails.

4. OUR APPROACH

This section first shows the basic idea of our solution to the problems, and then gives the algorithm $\mathcal{G}^{\mathrm{C}}$ for Featherweight Java for Relaxation with covariant return types (FJR$^{\mathrm{C}}$) to generate subtyping constraints from a given expression and a type environment. The algorithm is a small extension to the constraint generation algorithm $\mathcal{G}$ for FJR. Although our algorithm does not support generics, it would be easy if the base calculus FJR$^{\mathrm{C}}$ is extended to support it.

4.1 Basic idea

The basic idea of our solution is to extend the consistency checking rules so that they check the usage of the return values not only from the target join point but also the method calls that uses values *derived* from it. The definition of derived values is given below. If any inconsistencies are found, the advice is rejected.

Let `v` and `w` be values. We say that a method `m` *directly derives* `v` from `w` if `w` is the return value from a method call `v.m`. We also say that `w` is *derived* from `v` if

$$\mathcal{G}^{\mathrm{C}}(\Gamma, \mathtt{x}) = (\emptyset, \Gamma(\mathtt{x}))$$

$$\begin{array}{l}
\mathcal{G}^{\mathrm{C}}(\Gamma, \mathtt{let\ x\ =\ e_1\ in\ e_2}) = \\
\quad \mathbf{let}\ (\mathcal{R}^{\mathrm{C}}_1, \mathtt{U}_1) = \mathcal{G}^{\mathrm{C}}(\Gamma, \mathtt{e}_1)\ \mathbf{in} \\
\quad \mathbf{let}\ (\mathcal{R}^{\mathrm{C}}_2, \mathtt{U}_2) = \mathcal{G}^{\mathrm{C}}((\Gamma, \mathtt{x}\mathtt{:}\mathtt{U}_1), \mathtt{e}_2)\ \mathbf{in} \\
\quad (\mathcal{R}^{\mathrm{C}}_1 \cup \mathcal{R}^{\mathrm{C}}_2, \mathtt{U}_2)
\end{array}$$

$$\begin{array}{l}
\mathcal{G}^{\mathrm{C}}(\Gamma, \mathtt{e}_0\mathtt{.m(}\mathtt{e}_1\mathtt{,}\ \cdots\ \mathtt{,}\mathtt{e}_n\mathtt{)}) = \\
\quad \mathbf{let}\ (\mathcal{R}^{\mathrm{C}}_0, \mathtt{U}_0) = \mathcal{G}^{\mathrm{C}}(\Gamma, \mathtt{e}_0)\ \mathbf{in} \\
\quad \mathbf{let}\ (\mathcal{R}^{\mathrm{C}}_1, \mathtt{U}_1) = \mathcal{G}^{\mathrm{C}}(\Gamma, \mathtt{e}_1)\ \mathbf{in} \\
\qquad\qquad \vdots \\
\quad \mathbf{let}\ (\mathcal{R}^{\mathrm{C}}_n, \mathtt{U}_n) = \mathcal{G}^{\mathrm{C}}(\Gamma, \mathtt{e}_n)\ \mathbf{in} \\
\quad \mathbf{let}\ \overline{\mathtt{T}}\to\mathtt{T} = \mathit{mtype}(\mathtt{m}, \mathit{typeOf}(\mathtt{e}_0))\ \mathbf{in} \\
\quad \mathbf{let}\ \mathtt{V} = \bigcup \mathit{mdeftypes}(\mathtt{m}, \mathit{typeOf}(\mathtt{e}_0))\ \mathbf{in} \\
\quad (\mathcal{R}^{\mathrm{C}}_0 \cup \mathcal{R}^{\mathrm{C}}_1 \cup \cdots \cup \mathcal{R}^{\mathrm{C}}_n \cup \{\overline{\mathtt{U}}\ \mathtt{<:}\ \overline{\mathtt{T}}\} \\
\qquad \cup \{\mathtt{U}_0\ \mathtt{<:}\ \mathtt{X}, \mathtt{X}\ \mathtt{<:}\ \mathtt{V}, \lambda s.\mathit{mrtype}(\mathtt{m}, s\mathtt{X})\ \mathtt{<:}\ \mathtt{Y}\}, \\
\quad \mathtt{Y}) \\
\quad \text{(for fresh } \mathtt{X} \text{ and } \mathtt{Y}\text{)}
\end{array}$$

$$\mathcal{G}^{\mathrm{C}}(\Gamma, \mathtt{new\ C()}) = (\emptyset, \mathtt{C})$$

$$\begin{array}{l}
\mathcal{G}^{\mathrm{C}}(\Gamma, \mathtt{(?e_1:e_2)}) = \\
\quad \mathbf{let}\ (\mathcal{R}^{\mathrm{C}}_1, \mathtt{U}_1) = \mathcal{G}^{\mathrm{C}}(\Gamma, \mathtt{e}_1)\ \mathbf{in} \\
\quad \mathbf{let}\ (\mathcal{R}^{\mathrm{C}}_2, \mathtt{U}_2) = \mathcal{G}^{\mathrm{C}}(\Gamma, \mathtt{e}_2)\ \mathbf{in} \\
\quad (\mathcal{R}^{\mathrm{C}}_1 \cup \mathcal{R}^{\mathrm{C}}_2, \mathtt{U}_1 \cup \mathtt{U}_2)
\end{array}$$

Figure 4: Modified constraint generation algorithm

- some method `m` directly derives `w` from `v`, or
- `w` is derived from `v'` and `v'` is derived from `v` for some value `v'`.

In Figure 2, the return value from the join point `new BigIntegerStream()` at line 12 is assigned to `s`. We use the variable names to denote the return values for simplicity. `i` and `absi` are derived from `s` because `i` is directly derived from `s` and `absi` is directly derived from `i`.

Our extended rules check whether the return type of the method that directly derives `i` (resp. `absi`) and the operations that use `i` (resp. `absi`) are consistent if `s` is an `IntegerStream` object. The operation `s.get()` directly derives `i` and its return type is `Object`. `i.abs()` at line 12 uses `i` and its signature is `BigInteger BigInteger.abs()`, which can be no more relaxed and inconsistent with `Object`. Hence the rules rejects the around advice (lines 29–31 in Figure 1).

4.2 Constraint generation algorithm

We design an algorithm for the extended consistency checking rules on a small object-oriented language called Featherweight Java for Relaxation with covariant return types (FJR$^{\mathrm{C}}$), which is a simple extension to FJR [8]. In this section we first give the syntax rules of FJR$^{\mathrm{C}}$, which is the same to the ones of FJR, and the typing rules that are modified to support covariant return types. Then we give the algorithm $\mathcal{G}^{\mathrm{C}}$ and explain it through a simple example. Proving its formal correctness is not completed yet, which is left for our future work.

The syntax rules of FJR$^{\mathrm{C}}$are the same to the ones of FJR:

$$\begin{array}{rcl}
\texttt{CL} & ::= & \texttt{class C extends C implements } \overline{\texttt{I}} \texttt{ \{ } \overline{\texttt{M}} \texttt{ \}} \\
\texttt{M} & ::= & \texttt{T m(}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{) \{ return e; \}} \\
\texttt{IF} & ::= & \texttt{interface I \{ } \overline{\mathcal{S}} \texttt{ \}} \\
\mathcal{S} & ::= & \texttt{T m(}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{);} \\
\texttt{e} & ::= & \texttt{x} \mid \texttt{e.m(}\overline{\texttt{e}}\texttt{)} \mid \texttt{new C()} \\
 & & \mid \texttt{let x = e in e} \mid \texttt{(?e:e)} \\
\texttt{S,T} & ::= & \texttt{C} \mid \texttt{I} \\
\texttt{U,V} & ::= & \texttt{T} \mid \texttt{U} \cup \texttt{U}
\end{array}$$

An overline denotes a sequence, e.g., $\overline{\texttt{x}}$ is shorthand for $\texttt{x}_1,\ldots,\texttt{x}_n$. The metavariable `C` ranges over class names; `I` ranges over interface names; `m` ranges over method names; and `x` and `y` range over variables, which include the special variable `this`.

`CL` is a class declaration, consisting of its name, a superclass name, interface names that it implements, and methods $\overline{\texttt{M}}$; `IF` is an interface declaration, consisting of its name and method headers $\overline{\mathcal{S}}$.

The syntax of expressions includes `let` expressions to illustrate the cases when a value returned from around advice is used as values of different types. `let` is the only binding construct of an expression and the variable `x` in `let x =` $\texttt{e}_1$ `in` $\texttt{e}_2$ is bound in $\texttt{e}_2$. It also includes non-deterministic choice `(?e:e)` to handle the cases when a variable contains values of different types.

`S` and `T` stand for simple types, i.e., class and interface names, and will be used for types written down in classes and interfaces. `U` and `V` stand for union types. For example, a local variable of type $\texttt{C} \cup \texttt{D}$ may point to either an object of class `C` or that of `D`.

To support covariant return types, we need to change the typing rule T-CLASS, the predicate *override* and the constraint generation algorithm $\mathcal{G}$ to allow each overriding method to have a return type that is a subtype of the one of the method in its superclass and interfaces.

The modified typing rule T-CLASS is given as follows:

$$\frac{\begin{array}{l}\forall \texttt{m}, \texttt{I} \in \overline{\texttt{I}}. \\ \left\{\begin{array}{l} (\mathit{mtype}(\texttt{m},\texttt{I}) = \overline{\texttt{T}}\to\texttt{T}_0) \Longrightarrow (\mathit{mtype}^C(\texttt{m},\texttt{C}) = \overline{\texttt{T}}\to\texttt{S}_0) \\ \text{and } \texttt{S}_0 \texttt{<:} \texttt{T}_0 \end{array}\right\} \\ \qquad\qquad\qquad \overline{\texttt{M}}\ \texttt{OK IN C}\end{array}}{\texttt{class C extends D implements } \overline{\texttt{I}} \texttt{ \{ } \overline{\texttt{M}} \texttt{ \} OK}} \quad \text{(T-CLASS)}$$

where $\mathit{mtype}(\texttt{m},\texttt{I})$ and $\mathit{mtype}^C(\texttt{m},\texttt{C})$ are the functions that return It defines `C` is well-typed if all methods are well typed and all methods declared in $\overline{\texttt{I}}$ are implemented in `C` with signature that has a smaller return type.

The predicate $\mathit{override}(\texttt{m},\texttt{C},\overline{\texttt{T}}\to\texttt{T}_0)$, which checks whether `m` is correctly overrides the method of the same name in `C`, is modified similarly as follows:

$$\frac{(\mathit{mtype}(\texttt{m},\texttt{C}) = \overline{\texttt{S}}\to\texttt{S}_0) \Longrightarrow \overline{\texttt{S}} = \overline{\texttt{T}} \text{ and } \texttt{S}_0 \texttt{<:} \texttt{T}_0}{\mathit{override}(\texttt{m},\texttt{C},\overline{\texttt{T}}\to\texttt{T}_0)}$$

The modified constraint generation algorithm $\mathcal{G}^{\mathrm{C}}$ (Figure 4) takes a type environment Γ and an expression `e` and returns a set $\mathcal{R}^{\mathrm{C}}$ of extended subtyping constraints and a type `U`. An extended subtyping constraint is an inequality of the form `U <: V` or $\lambda s.\mathit{mrtype}(\texttt{m}, s\texttt{X})$ `<: U` where `U` and `V` range over either simple types or variables `X` and `Y`, $\mathit{mrtype}(\texttt{m},\texttt{T})$ returns the return type of the method `m` in the simple type `T`, and $\lambda s.\mathit{mrtype}(\texttt{m}, s\texttt{X})$ is a function that takes a substitution S of simple types $\overline{\texttt{T}}$ for type variables $\overline{\texttt{X}}$ and returns a simple type $\mathit{mrtype}(\texttt{m}, S\texttt{X})$. $\mathit{typeOf}(\texttt{e})$ denotes the simple type of a receiver `e` of a method invocation. $\mathit{mdeftypes}(\texttt{m},\texttt{T})$ collects the set of `T`'s supertypes that define `m`.

```
1 Object m(){
2   return
3     let s = (?new BigIntegerStream():
4             new IntegerStream())
5     in let i = s.get() BigInteger BigIntegerStream.get()
6     in let absi = i.abs()  BigInteger BigInteger.abs()
7     in new Object();
8 }
```

Figure 5: An example code of FJR$^{\mathrm{C}}$

The case for method invocations is different from the original constraint generation algorithm $\mathcal{G}$. The type variable `X` stands for the receiver type, which has to be a supertype of the expression $\texttt{e}_0$. `X <: V` guarantees that the receiver type has method `m` whose argument types are $\overline{\texttt{T}}$. The type variable `Y` stands for the return type, which depends on the receiver type. $\lambda s.\mathit{mrtype}(\texttt{m}, s\texttt{X})$ `<: Y` represents this fact.

Example.

The method `m` in Figure 2 can be written in FJR$^{\mathrm{C}}$ as Figure 5. The return type is changed from `void` to `Object` because FJR$^{\mathrm{C}}$ does not have it. The around advice is woven manually to lines 3–4 by using non-deterministic choice.

Our algorithm $\mathcal{G}^{\mathrm{C}}$ correctly rejects the program as follows. Applying $\mathcal{G}^{\mathrm{C}}$ to lines 3–4, we get to know that the type of `s` is $\texttt{BigInteger}\cup\texttt{Integer}$. At line 5, $\mathcal{G}^{\mathrm{C}}$ generates the constraints:

$$\left\{\begin{array}{c} \texttt{BigIntegerStream} \cup \texttt{IntegerStream} \texttt{ <: } \texttt{X}_1, \\ \texttt{X}_1 \texttt{ <: } \texttt{BigIntegerStream} \cup \texttt{Stream}, \\ \lambda s.\mathit{mrtype}(\texttt{get}, s\texttt{X}) \texttt{<:} \texttt{Y}_1 \end{array}\right\}$$

Because $\texttt{BigIntegerStream}\cup\texttt{Stream}$ can be reduced to `Stream`, `Stream` is the only candidate for $\texttt{X}_1$. We can also reduce the constraint for $\texttt{Y}_1$ and get $\texttt{Object<:Y}_1$, which indicates that $\texttt{Y}_1$ must be `Object`.

Evaluating $\mathcal{G}^{\mathrm{C}}$ on `i.abs()` at line 6, we get the constraint:

$$\left\{\ \texttt{Y}_1 \texttt{ <: } \texttt{X}_2,\quad \texttt{X}_2 \texttt{ <: } \texttt{BigInteger},\quad \lambda s.\mathit{mrtype}(\texttt{abs}, s\texttt{X}_2) \texttt{<:} \texttt{Y}_2\ \right\}$$

It is not satisfiable because there is no simple type such that `Object <:` $\texttt{X}_2$ `<: BigInteger`.

5. RELATED WORK

Featherweight Aspect GJ (FAGJ) [6] is a small calculus based on Featherweight GJ [5], which supports covariant return types and generics. Its focus is on studying the incorporation of generic types in AspectJ family of languages, i.e., the authors discuss about typeability and type safety of aspect-oriented programs with generics for the case when the information about type parameters is not available (as in Java bytecode) as well as when it is (as in source code).

StrongAspectJ [3] is another calculus based on Featherweight Java, which focuses on improving generality of advice in a type-safe manner. It supports covariant return types and generics, as FAGJ.

Our study can be seen as a first attempt to connect these work and the type relaxed weaving.

6. CONCLUSIONS AND FUTURE WORK

In order to support the Java 5 features such as the covariant return types and the generics, the type relaxed weaving should be extended to check, in addition to the type usages of the return value from a target join point, those of the derived values from the return value. The additional checks are straightforwardly incorporated into the type relaxed weaving by slightly modifying a rule for overriding a method.

We also gave an extended constraint generation algorithm for Featherweight Java for Relaxation with covariant return types. Although the paper does not include the formalization of the generics support, we presume that no special extension is needed with respect to the consistency checks of the derived values.

As well as proving the correctness of our constraint generation algorithm and implementing the compiler, generics is left for our future work. An interesting technical challenge is to find a type parameters from an object creation expression in Java bytecode, which employ the type-erasure strategy for generics.

7. REFERENCES

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

[2] Bruno De Fraine, Erik Ernst, and Mario Südholt. Essential AOP: the A calculus. In *Proceedings of ECOOP'10*, pages 101–125, 2010.

[3] Bruno De Fraine, Mario Südholt, and Viviane Jonckers. StrongAspectJ: Flexible and safe pointcut/advice bindings. In *Proceedings of AOSD'08*, pages 60–71, 2008.

[4] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In *Proceedings of AOSD'04*, pages 26–35, 2004.

[5] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *TOPLAS*, 23(3):396–450, 2001.

[6] Radha Jagadeesan, Alan Jeffrey, and James Riely. Typed parametric polymorphism for aspects. *Science of Computer Programming*, 63(3):267–296, 2006.

[7] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of ECOOP'01*, pages 327–353, 2001.

[8] Hidehiko Masuhara, Atsushi Igarashi, and Manabu Toyama. Type relaxed weaving. In *Proceedings of AOSD'10*, pages 121–132, 2010.

[9] Hidehiko Masuhara and Gregor Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *Proceedings of ECOOP'03*, pages 2–28, 2003.

A Semantics for Execution Levels with Exceptions

Ismael Figueroa*
PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile – Santiago, Chile
ifiguero@dcc.uchile.cl

Éric Tanter†
PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile – Santiago, Chile
etanter@dcc.uchile.cl

ABSTRACT

Aspect-oriented languages are usually formulated as an extension to existing languages, without paying any special attention to the underlying exception handling mechanisms. Consequently, aspect exceptions and handlers are no different than base exceptions and handlers. Conflation between aspect and base exceptions and handlers may inadvertently trigger execution of unintended handlers, changing the expected program behavior: aspect exceptions are accidentally caught by base handlers or vice-versa. Programmers cannot state the desired interaction between aspect and base exceptions and handlers. Specific instances of this issue have been identified by others researchers. We distill the essence of the problem and designate it as the *exception conflation problem*. Consequently, we propose a semantics for an aspect-oriented language with execution levels and an exception handling mechanism that solves the exception conflation problem. By default, the language ensures there is no interaction between base and aspect exceptions and handlers, and provides level-shifting operators to flexibly specify interaction between them when required. We illustrate the benefits of our proposal with a representative set of examples.

Categories and Subject Descriptors: D.3.3 [Programming Languages]: Language Constructs and Features

General Terms: Languages, Design

Keywords: Exception handling, aspect-oriented programming, exception conflation, execution levels

1. INTRODUCTION

Aspect-oriented languages aim at modularizing crosscutting concerns. Well-known aspect-oriented languages, like AspectJ, define crosscutting behavior using pointcuts and advices: join points are interesting events raised during program execution, pointcuts are predicates over join points to determine the application of advice. Advice is code that executes after, before or instead of the computation represented by a join point. An aspect is an abstraction to specify crosscutting behavior.

```
class A {
  public void foo() {
    Integer configValue;
    try { configValue = getConfiguration();
    } catch(Exception ex) { configValue = DEFAULT}}
}
aspect Logging {
  Object around() : call(Integer getConfiguration()) {
    logger.append("Calling getConfiguration"); // FileNotFoundException
    return proceed();}
}
```

Listing 1: Base handler catches aspect exception

Exceptions are a mechanism to deal with abnormal states of computation in a uniform and modular way by a lexical separation between normal and error handling code. When an abnormal situation ocurrs, an exception is thrown and then propagates dynamically through the call stack in search for a suitable handler. A handler is a special code section which executes receiving an exception as parameter. If a handler is not found the exception is uncaught, usually aborting program execution.

Subtle interactions between aspect and base exceptions and handlers may arise. Aspect exceptions may be inadvertently handled by base handlers. Conversely, base exceptions may be caught by aspect handlers. For example, consider the AspectJ code of Listing 1. The foo method calls getConfiguration to set configValue. In case of failure, a default value is used. A Logging aspect advises around the getConfiguration method. If the logger object cannot find the log file it shall fail and throw a FileNotFoundException, which is caught by the base handler. Thus, the default value is used because the aspect failed, even in cases where getConfiguration would have returned normally.

This situation arises because the exception handling mechanism merges aspect and base handlers and exceptions in a flat structure. We call this situation the *exception conflation problem*. The exception conflation problem is a generalization of the *Late Binding Handler Pattern* of Coelho *et al.*, which is described as: "...happens when an aspect is created to handle an exception, but the aspect intercepts a point in the program execution where the exception to be caught was already caught by a handler in the method call chain that connects the exception signaler to the aspect handler" [4].

In this paper we propose a semantics for an aspect-oriented language that discriminates aspect and base exceptions using *execution levels*, extending Tanter's original proposal [10]. The language ensures that by default there is no interference between aspect and base exceptions and provides level-shifting operators to flexibly

*Funded by a CONICYT Doctoral Scolarship.

†Partially funded by FONDECYT Project 1110051.

FOAL'11, March 21, 2011, Pernambuco, Brazil.
Copyright 2011 ACM 978-1-4503-0644-7/11/03 ...$10.00.

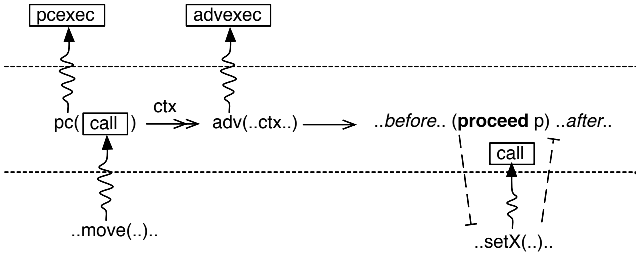


Figure 1: Execution levels in action: pointcut and advice are evaluated at level 1, *proceed* **goes back to level 0 (from [10]).**

specify their interactions when required. We illustrate the benefits of our proposal showing representative examples of interaction between aspect and base exceptions and handlers.

The rest of this paper is structured as follows: Section 2 recalls the notion of execution levels, Section 3 defines the semantics of the proposed language, Section 4 shows the applications of this language to exception handling issues, Section 5 discusses related work, and Section 6 concludes.

2. EXECUTION LEVELS IN A NUTSHELL

We first summarize the proposal of execution levels. An aspect observes the execution of a program through its pointcuts, and affects it with its advice. An advice is a piece of code, and therefore its execution also produces join points. Similarly, pointcuts as well can produce join points. For instance, in AspectJ, one can use an *if* pointcut designator to specify an arbitrary Java expression that ought to be true for the pointcut to match. The evaluation of this expression is a computation that produces join points. In higher-order aspect languages like AspectScript [12] and others, all pointcuts and advice are standard functions, whose application and evaluation produce join points as well.

The fact that aspectual computation produces join points raises the crucial issue of the *visibility* of these join points. In most languages, aspectual computation is visible to all aspects—including themselves. This of course opens the door to infinite regression and unwanted interference between aspects. These issues are typically addressed with ad-hoc checks (*e.g.* using **!within** and **cflow** checks in AspectJ) or primitive mechanisms (like AspectScheme's **app/prim**). However, all these approaches eventually fall short for they fail to address the fundamental problem, which is that of *conflating* levels that ought to be kept separate [2].

In order to address the above issue, Tanter proposed execution levels for AOP [10]. A program computation is structured in *levels*. Computation happening at level 0 produces join points observable at level 1 only. Aspects are *deployed* at a particular level, and observe only join points at that level. This means that an aspect deployed at level 1 only observes join points produced by level-0 computation. In turn, the computation of an aspect (*i.e.* the evaluation of its pointcuts and advice) is reified as join points visible at the level immediately above: therefore, the activity of an aspect standing at level 1 produces join points at level 2.

An aspect that acts *around* a join point can invoke the original computation. For instance, in AspectJ, this is done by invoking *proceed* in the advice body. The original computation ought to run at the same level at which it originated![1] In order to address this issue, it is important to remember that when several aspects match the same join point, the corresponding advice are chained, such that calling **proceed** in advice k triggers advice $k+1$. Therefore, the semantics of execution levels guarantees that the *last call* to **proceed** in a chain of advice triggers the original computation at the lower original level.

This is shown in Figure 1. A call to a `move` method in the program produces a call join point (at level 1), against which a pointcut `pc` is evaluated. The evaluation of `pc` produces join points at level 2. If the pointcut matches, it passes context information `ctx` to the advice. Advice execution produces join points at level 2, except for **proceed**: control goes back to level 0 to perform the original computation, then goes back to level 1 for the after part of the advice.

Level-shifting operators. The default semantics of execution levels treats aspects as a meta-level computation. However, in some cases, advice execution should be visible to aspects that observe base level execution. To reconcile both approaches, Tanter proposed explicit level-shifting operators: **up** and **down**. Shifting an expression using **up** or **down** moves its computation one level above or below, affecting the visibility of its join points. With these operators the programmer can specify the level at which computation is performed, according to specific needs.

Level-capturing functions. Certain applications require delayed execution of operations, for example: a prioritized command queue may accumulate a number of requests before executing them or a task scheduler may execute certain tasks at determined time intervals. In these cases, the operations may not be performed in the same execution thread in which they were declared, so the control flow checks to avoid regression problems fail. In addition, the execution may be performed at a different level than when it was postponed. To address this issue Tanter introduces level-capturing functions that keep track of the level at which they were declared. When a level-capturing function is executed, the execution level shifts to the function-captured level, and then shifts back afterwards.

By separating execution into levels, unwanted interactions between aspects are avoided. For instance, it becomes possible to reuse off-the-shelf dynamic analysis aspects and apply them to a given program, and/or aspects, with consistent semantics [11].

3. SEMANTICS

In this section we first recall the core syntax and semantics of Tanter's original proposal [10], which we then extend with exception handling. The complete semantics for the original language and the exception handling extensions are defined using PLT Redex, a domain-specific language for specifying reduction semantics [5]. The complete semantics implementation, along an executable test suite, and examples shown in Section 4 are available at `http://pleiad.cl/research/scope`.

3.1 Original Semantics

Tanter's proposal defines a simple Scheme-like language with higher-order aspects, and execution levels as described in Section 2. The language has booleans, numbers and lists, primitives functions to operate on these, and an *internal* `app/prim` operator to apply functions without generating join points [10]. Figure 2 shows the core syntax and reduction rules of the language. The user-visible expressions are values, identifiers, **if** statements, multi-arity function application, and aspect deployment. These are shown in **bold** font. The other expressions shown in the figure are related to level-shifting operations, and are shown with `typewriter` font.

A reduction relation $\hookrightarrow$ describes the operational semantics of

[1] This issue is precisely why using control flow checks in AspectJ in order to discriminate advice computation is actually flawed. See [10] for more details.

$$
\begin{array}{llll}
\mathit{Value} & v & ::= & (\lambda(x \cdots)\, e) \mid (\lambda^{\bullet}(x \cdots)\, e) \\
 & & & \mid n \mid \#t \mid \#f \\
 & & & \mid (\textbf{list}\, v \cdots) \mid \mathit{prim} \mid \textit{unspecified} \\
 & \mathit{prim} & ::= & \textbf{deploy} \mid \textbf{list} \mid \textbf{cons} \mid \textbf{car} \mid \textbf{cdr} \\
 & & & \mid \textbf{empty?} \mid \textbf{eq?} \mid + \mid - \mid \ldots \\
\mathit{Expr} & e & ::= & v \mid x \mid (e\, e \cdots) \mid (\textbf{if}\, e\, e\, e) \\
 & & & \mid (\textbf{up}\, e) \mid (\textbf{down}\, e) \\
 & & & \mid (\texttt{in-up}\, e) \mid (\texttt{in-down}\, e) \\
 & & & \mid (\texttt{in-shift}(e)) \\
 & & & \mid (\texttt{app/prim}\, e\, e \cdots) \\
 & v & \in & \mathscr{V}\text{, the set of values} \\
 & n & \in & \mathscr{N}\text{, the set of numbers} \\
 & \mathit{list} & \in & \mathscr{L}\text{, the set of lists} \\
 & x & \in & \mathscr{X}\text{, the set of variable names} \\
 & e & \in & \mathscr{E}\text{, the set of expressions} \\
\mathit{EvalCtx} & E & ::= & [\,] \mid (v \cdots\, E\, e \cdots) \mid (\textbf{if}\, E\, e\, e) \\
 & & & \mid (\textbf{up}\, E) \mid (\textbf{down}\, E) \\
 & & & \mid (\texttt{in-up}\, E) \mid (\texttt{in-down}\, E) \\
 & & & \mid (\texttt{in-shift}(l)\, E) \\
 & & & \mid (\texttt{app/prim}\, v \cdots\, E\, e \cdots)
\end{array}
$$

$$
\begin{array}{ll}
\langle l, J, E[(\textbf{up}\, e)] \rangle \hookrightarrow \langle l+1, J, E[(\texttt{in-up}\, e)] \rangle & \textsc{InUp} \\
\langle l, J, E[(\texttt{in-up}\, v)] \rangle \hookrightarrow \langle l-1, J, E[v] \rangle & \textsc{OutUp} \\
\langle l, J, E[(\textbf{down}\, e)] \rangle \hookrightarrow \langle l-1, J, E[(\texttt{in-down}\, e)] \rangle & \textsc{InDwn} \\
\langle l, J, E[(\texttt{in-down}\, v)] \rangle \hookrightarrow \langle l+1, J, E[v] \rangle & \textsc{OutDwn} \\
\langle l, J, E[(\lambda^{\bullet}(x \cdots)\, e) \rangle & \textsc{Capture} \\
\qquad \hookrightarrow \langle l, J, E[(\lambda^{l}(x \cdots)\, e)] \rangle & \\
\langle l, J, E[(\texttt{app/prim}\, (\lambda(x \cdots)\, e)\, v \cdots)] \rangle & \textsc{AppPrim} \\
\qquad \hookrightarrow \langle l, J, E[e\{v \cdots / x \cdots\}] \rangle & \\
\langle l_1, J, E[(\texttt{app/prim}\, (\lambda^{l_2}(x \cdots)\, e)\, v \cdots)] \rangle & \textsc{AppShift} \\
\qquad \hookrightarrow \langle l_2, J, E[(\texttt{in-shift}(l_1)\, e\{v \cdots / x \cdots\})] \rangle & \\
\langle l_2, J, E[(\texttt{in-shift}(l_1)\, v) \rangle \hookrightarrow \langle l_1, J, E[v] \rangle & \textsc{Shift}
\end{array}
$$

Figure 2: Core syntax and reduction rules of the language.

our language using reduction steps. The relation $\hookrightarrow$ is defined as follows[2]: $\hookrightarrow : \mathscr{L} \times \mathscr{J} \times \mathscr{E} \rightarrow \mathscr{L} \times \mathscr{J} \times \mathscr{E}$

An evaluation context consists of an execution level $l \in \mathscr{L}$, a join point stack $J \in \mathscr{J}$ and an expression $e \in \mathscr{E}$. The reduction relation takes a level, a join point stack, and an expression and maps this to a new evaluation step. Join point definition, aspect deployment and the weaving mechanism is described in [10]. The exception handling extension does not alter these definitions nor any reduction rules of the original language.

Primitive application. The language features a primitive function application, `app/prim`, that does not generate join points. It performs a simple β_v reduction of the expression. This mechanism is required for tasks such as the initial application of the composed advice chain (and its recursive calls), and to perform the original computation when all aspects (if any) have proceeded. This is shown in rule AppPrim.

Level-shifting operators. The **up** (**down**) level-shifting operator embeds its inner expression in an `in-up` (`in-down`) expression, which increases (lowers) the current execution level. When the embedded expression is reduced to a value, the execution level is decreased (increased) to the original level. This is specified by the rules InUp, InDwn, OutUp, OutDwn. Aspect weaving ensures that pointcuts and advices are evaluated with **up**, and that the last **proceed** in the chain is evaluated with **down** [10].

Level-capturing functions. Level-capturing functions are defined using $\lambda^{\bullet}$. A function value bound to level l is denoted λ^{l}. Rule Capture shows that when a level-capturing function is defined, it is tagged with the current execution level. Rule AppShift shows that when a level-capturing function is applied, it embeds the reduction of the application in an `in-shift` expression. By rule Shift, when the expression reduces to a value, the execution level shifts back to the level embedded in the `in-shift` form.

Aspect deployment. Aspects can be dynamically deployed. The **deploy** expression takes a pointcut and an advice and adds the aspect to the current aspect environment. To deploy aspects of aspects we use the level-shifting operators to shift the level at which the **deploy** expression is evaluated. The complete rules for aspect deployment are shown in Tanter's proposal [10], and are not changed in our proposal.

3.2 Exception Handling Semantics

We introduce a standard exception handling mechanism [9], using the extensions shown in Figure 3. We extend the user-visible syntax with the **raise** and **try-with** expressions and their respective evaluation contexts. We also define an *Exception* normal form annotated with the level at the which the exception was raised. Then we add reduction rules to specify the semantics of the **raise** and **try-with** expression. In essence, our proposal consists in tagging exceptions and handlers with their respective execution level; then, an exception is only caught by a suitable handler if they are both bound at the same level.

Safe default. A **try** e **with** e_h expression contains a protected expression e and a handler expression e_h. If an exception is raised during the reduction of e, e_h is evaluated, and the resulting function is applied to the exception, *only if* the level of the exception matches the level of the handler. This default semantics ensures that there is no interaction between aspect and base exceptions and handler. For instance, if there are no explicit level shifts, the computation of e happens at level 0. If an exception is thrown by the base code, it will be a level-0 exception, which will therefore be caught by the handler. Conversely, if the exception is thrown in an aspect (at level 1), the handler will not catch the exception.

Certainly, there are situations where interaction is desired. To this end, we exploit the fact that the handler is an expression to be evaluated to install a handler in an upper (or lower) level using level-capturing functions and level shifting operators. This is illustrated in Section 4.

Raising exceptions. The **raise** expression signals an exception that embeds a value to carry information to the handler. Rule Raise-Capture creates a tagged exception, which holds the execution level at which the exception is raised. An exception bound at level l is denoted ($\textbf{raise}^{l}\, v$).

Exception propagation. The rule Reraise deals with exception propagation in nested **raise** expressions. An exception propagates through the `in-up` and `in-down` expressions maintaining its tagged level. For simplicity we omit the propagation rules here. They are present in the downloadable complete semantics specification.

Handling exceptions. As reflected by rule TryV, if the protected expression of a **try-with** reduces to a value, the whole **try-with** expression reduces to that value. Otherwise, the reduction is determined by one of these rules: HndEx1, HndProp1, HndEx2 and HndProp2. It is important to observe that the **try** ex **with** E execution context ensures that the handler expression is only eval-

[2] The complete semantics we provide includes the aspect environment in the reduction rules, omitted here for simplicity.

$$
\begin{array}{llll}
\mathit{Expr} & e & ::= & \cdots \mid (\textbf{raise}\ e) \mid (\textbf{try}\ e\ \textbf{with}\ e) \\
\mathit{Exception} & ex & ::= & (\textbf{raise}^{l}\ v) \\
\mathit{EvalCtx} & E & ::= & \cdots \mid (\textbf{raise}\ E) \mid (\textbf{try}\ E\ \textbf{with}\ e) \\
 & & & \mid (\textbf{try}\ ex\ \textbf{with}\ E)
\end{array}
$$

$$\langle l, J, E[(\textbf{raise}\ v)]\rangle \qquad \textsc{RaiseCapture}$$
$$\hookrightarrow \langle l, J, E[(\textbf{raise}^{l}\ v)]\rangle$$

$$\langle l, J, E[(\textbf{raise}\ (\textbf{raise}^{l_1}\ e))]\rangle \qquad \textsc{Reraise}$$
$$\hookrightarrow \langle l, J, E[(\textbf{raise}^{l_1}\ e)]\rangle$$

$$\langle l, J, E[(\textbf{try}\ v\ \textbf{with}\ e)]\rangle \qquad \textsc{TryV}$$
$$\hookrightarrow \langle l_1, J, E[v]\rangle$$

$$\langle l, J, E[(\textbf{try}\ (\textbf{raise}^{l}\ v)\ \textbf{with}\ (\lambda(x\cdots)\ e)]\rangle \qquad \textsc{HndEx1}$$
$$\hookrightarrow \langle l, J, E[((\lambda(x\cdots)\ e)\ v)]\rangle$$

$$\langle l, J, E[(\textbf{try}\ (\textbf{raise}^{l_1}\ v)\ \textbf{with}\ (\lambda(x\cdots)\ e))]\rangle \qquad \textsc{HndProp1}$$
$$\hookrightarrow \langle l, J, E[(\textbf{raise}^{l_1}\ v)]\rangle \text{ where } l_1 \neq l$$

$$\langle l, J, E[(\textbf{try}\ (\textbf{raise}^{l_1}\ v)\ \textbf{with}\ (\lambda^{l_1}(x\cdots)\ e)]\rangle \qquad \textsc{HndEx2}$$
$$\hookrightarrow \langle l_1, J, E[(\texttt{in-shift}(l)\ ((\lambda^{l_1}(x\cdots)\ e)\ v))]\rangle$$

$$\langle l, J, E[(\textbf{try}\ (\textbf{raise}^{l_1}\ v)\ \textbf{with}\ (\lambda^{l_2}(x\cdots)\ e))]\rangle \qquad \textsc{HndProp2}$$
$$\hookrightarrow \langle l, J, E[(\textbf{raise}^{l_1}\ v)]\rangle \text{ where } l_1 \neq l_2$$

Figure 3: Exception handling extensions.

uated when an exception is raised in the evaluation of the protected expression.

The HNDEX1 and HNDPROP1 rules deal with normal function handlers: if the exception level matches the current execution level, the handler is applied; else, the exception propagates in search of a suitable handler. Rules HNDEX2 and HNDPROP2 deal with level-capturing function handlers: regardless of the current execution level, if the exception and handler level match then the handler is applied shifting the current execution level to the level of the handler; otherwise the exception keeps its propagation.

4. APPLICATIONS

To illustrate the benefits of our proposal, in this section we show a set of four representative examples of interaction between aspect and base exceptions and handlers: no interference between base exceptions and aspect handlers (Listing 2); no interference between base handlers and aspect exceptions (Listing 3); an aspect handler catching a base level exception (Listing 4), and a base handler catching an aspect exception (Listing 5). These minimal examples show that the semantics proposed in Section 3.2 solve the exception conflation problem, while still enabling interesting programming patterns, like aspects explicitly handling base exceptions.

For all the examples we assume that the pointcut associated to the advice matches any function call. The advice is a function which receives at least two parameters: p is the call to proceed, and c is the return value of the pointcut application to the join point. We also assume that the aspect is bound at level 1, so it observes only joint points emitted from base function calls.

Additional parameters are required for each parameter of the advised function. In all the examples each function takes exactly one argument, so each advice has three parameters: p, c, and arg which holds the value of the advised function parameter.

In Listing 2 the function of the normal expression in the **try-with** generates a call join point at level 1; the pointcut matches the join point and the advice raises an exception at level 1. The exception propagates back to the **try-with** expression and as the handler is at level 0, the exception is not caught. In consequence, the base expression reduces to (**raise**^{1} #f).

```
;; Advice
(λ (p c arg) (raise #f))

;; Base code
(try ((λ (x) x) #t) with (λ (ex) ex))
```

Listing 2: By default there is no interference between base exceptions and aspect handlers. Base expression evaluates to (raise^1 #f).

In Listing 3 the base generates a call join point at level 0; when the advice calls the proceed function p, an exception is raised at level 0 (remember that the last call to proceed executes at the original level, see Section 2) and as the handler in the advice is at level 1 the exception is uncaught. Thus, the base expression reduces to (**raise**^{0} #f).

```
;; Advice
(λ (p c arg) (try (p arg) with (λ (ex) ex)))

;; Base code
((λ (x) (raise #f)) #f)
```

Listing 3: By default there is no interference between aspect exceptions and base handlers. Base expression evaluates to (raise^0 #f).

Listing 4 shows the same situation as in Listing 3, except for the handler in the advice code. In this case, we shift down the evaluation of a level-capturing function using the **down** level-shifting operator. This causes the handler function to be bound at level 0. When the call to proceed raises the exception bound at level 0, the handler catches it because they are both bound to the same level. Note that the handler execution is a function application that happens at the level the handler is bound. This application also generates a call join point at level 0, and the advice applies again, but in this case no exception is raised in the call to proceed. Hence the original base code expression reduces to #f.

```
;; Advice
(λ (p c arg) (try (p arg) with (down (λ• (ex) ex))))

;; Base code
((λ (x) (raise #f)) #f)
```

Listing 4: Using a level-capturing function and the down level-shifting operator to catch a base exception in an aspect handler. Base expression evaluates to #f.

Finally, Listing 5 shows the same situation as in Listing 2, but with a different handler. In this case we use the **up** level-shifting operator to bind the handler function at level 1. When the aspect exception propagates to the **try-with** expression, the handler catches the exception and executes at level 1. In this case the advice does not execute again because the call join generates at level 2, and the aspect does not *see* it.

```
;; Advice
(λ (p c arg) (raise #f))

;; Base code
(try ((λ (x) x) #t) with (up (λ• (ex) ex))
```

Listing 5: Using a level-capturing function and the up level-shifting operator to catch an aspect exception in a base handler. Base expression evaluates to #t.

For each example, the handler has a normal function or a level-capturing function. The dual situation in which the handler is of the other kind is omitted. In the examples of Listing 2 and Listing 3 the program evaluates to the same result using a level-capturing function. In the other two examples, using a non level-capturing function the handler level does not match the exception level so the exception propagates.

The examples show the key interactions between aspect and base handlers and exceptions, and the mechanism that our semantics provide to the programmer to specify the desired interaction. Other situations like aspects of aspects can be reduced to one of the examples.

5. RELATED WORK

The first approach dealing with exception handling as a crosscutting concern was the study done by Lippert and Lopes [8]. They refactored a Java business application framework, called *JWAM*, using an old version of AspectJ to separate normal code from error handling code, in order to promote independent evolution and reusability of each section. They significantly reduced the LOC in the framework, managed to reuse common exception handling code and described several limitations of the AspectJ version used.

Castor Filho *et al.* [7] studied the adequacy of AspectJ for modularizing and reusing exception-handling code. They studied five systems: four object-oriented and one aspect-oriented. The object-oriented systems had their exception-handling code refactored to exception aspects. They obtained quantitative results applying a metrics suite based on four quality attributes: separation of concerns, coupling, cohesion and conciseness, to the systems. They also obtained qualitative results, they discuss several issues like best practices for aspect-oriented exception handling, interaction between exception-handling aspects and other aspects and scalability issues. Their main conclusions are that the mere use of aspects to handle exceptions is not sufficient to improve the quality of software, a careful design from the early phases of development is required to properly aspectize exception handling. A follow-up study addresses different design scenarios to aspectize exception handling in object-oriented systems [6].

Coelho *et al.* [3] study the interaction between aspects and exceptions in three systems with a Java and AspectJ versions available. They categorized the exception paths in the systems and the most common handlers strategies. Using their own static analysis tool they compare the object-oriented and aspect-oriented versions of each system. Using this information they developed a catalogue of exception-handling bug patterns in aspect-oriented programs [4].

6. CONCLUSION AND FUTURE WORK

In this paper we showed that interaction between aspect and base exceptions and handlers is prone to unintended execution of handlers and a lack of flexibility for the programmer. This situation ocurrs because the exception handling mechanisms do not distinguish between aspect and base exceptions. As a solution, we designed and described the semantics of a higher-order aspect language with execution levels and exceptions. Our language by default ensures there is no interaction between aspect and base handlers and exceptions and provides level-shifting operators to specify the interaction between them. We then showed four representative examples of interaction between aspects and base exceptions and how our language solves the exception conflation problem by default, and still provides the necessary flexibility when needed.

To further illustrate the benefits of our proposal, we plan to extend the AspectJ implementation with execution levels described by Tanter *et al.* in [11] with our exception handling semantics. Using this implementation we will make case studies similar to the ones done by Coelho *et al.* in [3], as well as some others like using the Contract4J design-by-contract framework. Other issues to consider are the interactions between exception handling mechanisms and the type system, the contrasts between languages with checked and unchecked exceptions, such as Java and C#; and the presence of finally blocks.

7. REFERENCES

[1] *Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD 2010)*, Rennes and Saint Malo, France, Mar. 2010. ACM Press.

[2] S. Chiba, G. Kiczales, and J. Lamping. Avoiding confusion in metacircularity: The meta-helix. In *Proceedings of the 2nd International Symposium on Object Technologies for Advanced Software (ISOTAS'96)*, volume 1049 of *Lecture Notes in Computer Science*, pages 157–172. Springer-Verlag, 1996.

[3] R. Coelho, A. Rashid, A. Garcia, N. Cacho, U. Kulesza, A. Staa, and C. Lucena. Assessing the impact of aspects on exception flows: An exploratory study. In J. Vitek, editor, *Proceedings of the 22nd European Conference on Object-oriented Programming (ECOOP 2008)*, number 5142 in Lecture Notes in Computer Science, pages 207–234, Paphos, Cyprus, july 2008. Springer-Verlag.

[4] R. Coelho, A. Rashid, A. von Staa, J. Noble, U. Kulesza, and C. Lucena. A catalogue of bug patterns for exception handling in aspect-oriented programs. In *PLoP '08: Proceedings of the 15th Conference on Pattern Languages of Programs*, pages 1–13, New York, NY, USA, 2008. ACM.

[5] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

[6] F. Filho, A. Garcia, and C. Rubira. Extracting error handling to aspects: A cookbook. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pages 134 –143, Oct. 2007.

[7] F. C. Filho, N. Cacho, E. Figueiredo, R. Maranhão, A. Garcia, and C. M. F. Rubira. Exceptions and aspects: the devil is in the details. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, pages 152–162, New York, NY, USA, 2006. ACM.

[8] 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*, New York, NY, USA, 2000. ACM.

[9] B. C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[10] É. Tanter. Execution levels for aspect-oriented programming. In AOSD 2010 [1], pages 37–48. Best Paper Award.

[11] É. Tanter, P. Moret, W. Binder, and D. Ansaloni. Composition of dynamic analysis aspects. In *Proceedings of the 9th ACM SIGPLAN International Conference on Generative Programming and Component Engineering (GPCE 2010)*, pages 113–122, Eindhoven, The Netherlands, Oct. 2010. ACM Press.

[12] R. Toledo, P. Leger, and É. Tanter. AspectScript: Expressive aspects for the Web. In AOSD 2010 [1], pages 13–24.

ContextFJ:
A Minimal Core Calculus for Context-oriented Programming

Robert Hirschfeld
Hasso-Plattner-Institut Potsdam
hirschfeld@hpi.uni-potsdam.de

Atsushi Igarashi
Kyoto University
igarashi@kuis.kyoto-u.ac.jp

Hidehiko Masuhara
The University of Tokyo
masuhara@acm.org

Abstract

We develop a minimal core calculus called ContextFJ to model language mechanisms for context-oriented programming (COP). Unlike other formal models of COP, ContextFJ has a direct operational semantics that can serve as a precise description of the core of COP languages. We also discuss a simple type system that helps to prevent undefined methods from being accessed via `proceed`.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Language, Theory

Keywords Context-oriented programming, operational semantics

1. Introduction

Context-oriented programming (COP) is an approach to improving modularity of behavioral variations that depend on the dynamic context of the execution environment [7]. In traditional programming paradigms, such behavioral variations tend to be scattered over several modules, and system architectures that support their dynamic composition are often complicated.

Many COP extensions including those designed on top of Java [2], Smalltalk [6], Common Lisp [4] and JavaScript [10], are based on object-oriented programming languages and introduce *layers* of *partial methods* for defining and organizing behavioral variations, and *layer activation mechanisms* for layer selection and composition. A partial method in a layer is a method that can run before, after, or around the same (partial) method defined in a different layer or a class. A layer groups related partial methods and can be (de)activated at run-time. It so contributes to the specific behavior of a set of objects in response to messages sent and received.

In this paper, we report on our ongoing work on a formal model of core language features of COP. We present a small calculus called ContextFJ that is an extension of Featherweight Java (FJ) [8]. As the first step, we severely limit the supported language features in ContextFJ so as to make the calculus simple yet expressive enough to add more interesting features in future. In addition to the usual features of FJ, it supports around-type (i.e., overriding) partial methods, dynamic activation and deactivation of layers, and `proceed` and `super` calls. Aside from the Java features FJ already omits, ContextFJ does not (yet) support first-class layers that can be passed around via arguments or variables, stateful layers that allow to share state between partial methods or associated objects, and before and after methods.

We give a small-step reduction semantics to model the behavior of COP programs *directly* without using translation to a language without COP features. As far as we know, this is a first direct semantics of COP features. Such direct semantics can serve as a precise specification of the core of COP languages.

We also discuss a type system for ContextFJ. As usual, the task of a type system is to statically guarantee the absence of run-time field-not-found and method-not-found errors. However, since in COP the presence of a method definition in a given class may depend on whether a particular layer is activated or not, this task is much harder. As a starting point, we develop a simple but restrictive type system, which allows partial methods only for existing methods in classes. We state that this simple type system is sound; a full version of the paper, available at `http://www.sato.kuis.kyoto-u.ac.jp/~igarashi/papers/`, includes proofs.

The rest of the paper is organized as follows. We first start with reviewing the language mechanisms for COP in Section 2. Section 3 defines the syntax and operational semantics of ContextFJ and Section 4 defines a simple type system. We discuss related and future work in Section 5.

2. Language Constructs for COP

2.1 Partial Method Definitions and Layers

We briefly review basic constructs along with their usage. In our example, behavioral variations are expressed as partial method definitions and related method definitions are grouped in layers.

```
class Person {
  String name, residence, employer;
  Person(String _name, String _residence,
         String _employer) {
    name = _name; residence = _residence;
    employer = _employer;
  }
  String toString() { return "Name: " + name; }
  layer Contact {
    String toString() {
      return
        proceed() + "; Residence: " + residence;
  }}
  layer Employment {
    String toString() {
      return proceed() + "; Affil.: " + employer;
}}}
```

FOAL'11, March 21, 2011, Pernambuco, Brazil.

Class Person defines three fields name, residence, and employer (all of type String) that will be initialized during object creation. It also defines the toString() method to show object-specific information referred to in its fields.

The base definition incorporates the name field in the textual representation. It belongs to the so-called base layer and with that is effective for all instances of Person (and its subclasses).

The one refinement is implemented as a partial definition of the same method in class Person and associated with the Contact layer. (COP layers are usually used to group more than one partial method definition, but as an illustrating example for ContextFJ, having one layer holding on to one partial method definition will suffice.) Our partial definition of toString() appends information from the residence field that might be useful for further correspondence. It belongs to the Contact layer and is only effective if Contact is active.

The other refinement is associated with the Employment layer and differs from the second refinement in the field, now employer, that it deals with.

In our example, the partial definitions of toString() call the special method proceed() to invoke other partial definitions of toString() contributed by layers that were already active before the activation of the Contact or the Employment layer, or to invoke the base-level implementation of this method (here provided by class Person).

proceed(...) is similar to super as it allows for calling other behavior previously defined in the composition path: Whereas super changes the starting point of the method lookup to the superclass of the class the (partial) method was defined in, proceed(...) will try to find the next partial or base-level definition of the same method. If proceed(...) cannot find such a partial method in the current receiver class or the active layers associated with it, lookup continues in the superclass of the current lookup class. Lookup is statically guaranteed to succeed as we will see in Section 4.

2.2 Layer Activation and Deactivation

Layers are explicitly activated or deactivated using the with and without constructs respectively. In the following transcript, we show the application of theses constructs to an instance of class Person.

```
Person atsushi =
  new Person("Atsushi", "Kyoto", "Kyodai");
```

Printing our object to the standard output stream via println(...) with *no layers activated* leads to directly calling our base-level implementation that returns a description covering only the name of the object.

```
System.out.println(atsushi);
=> "Name: Atsushi"
```

However, if we put a with statement activating the Contact layer around this code, the same attempt to print out our object leads to first calling our partial definition of toString() contributed by the Contact layer which is responsible for providing contact information from the residence field, and then calling our base-level implementation providing the person's name.

```
with (Contact) { System.out.println(atsushi); }
=> "Name: Atsushi; Residence: Kyoto"
```

The nesting of with (or without) statements leads to nested layer activations, where "inner" layers gain precedence over "outer" ones.

```
with (Employment) {
  with (Contact) { System.out.println(atsushi); }
}
=> "Name: Atsushi; Residence: Kyoto; Affil.: Kyodai"
```

With that, the change of the order of with or without statements corresponds directly to the partial method definitions obtained by the method lookup.

```
with (Contact) {
  with (Employment) { System.out.println(atsushi); }
}
=> "Name: Atsushi; Affil.: Kyodai; Residence: Kyoto"
```

Previously activated layers using with can be deactivated by without and vice versa.

```
with (Contact) {
  without (Contact) { System.out.println(atsushi); }
}
=> "Name: Atsushi"
```

An attempt to deactivate a layer that is not active will not affect the current layer composition.

```
without (Contact) { System.out.println(atsushi); }
=> "Name: Atsushi"
```

As in most COP language extensions and also in ours, layer compositions are effective for the *dynamic extent* of the execution of the code block enclosed by their corresponding with or without statements[1].

3. Syntax and Semantics of ContextFJ

3.1 Syntax

Let metavariables C, D, and E range over class names; L over layer names; f and g over field names; m over method names; and x and y over variables, which contain a special variable this. The abstract syntax of ContextFJ is given as follows:

$$\begin{array}{lcll}
\texttt{CL} & ::= & \texttt{class C} \triangleleft \texttt{ C \{ } \overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{; K } \overline{\texttt{M}} \texttt{ \}} & (\textit{classes}) \\
\texttt{K} & ::= & & (\textit{constructors}) \\
 & & \texttt{C(}\overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{)\{ super(}\overline{\texttt{f}}\texttt{); this.}\overline{\texttt{f}}\texttt{ = }\overline{\texttt{f}}\texttt{; \}} & \\
\texttt{M} & ::= & \texttt{C m(}\overline{\texttt{C}}\ \overline{\texttt{x}}\texttt{)\{ return e; \}} & (\textit{methods}) \\
\texttt{e},\texttt{d} & ::= & \texttt{x} \mid \texttt{e.f} \mid \texttt{e.m(}\overline{\texttt{e}}\texttt{)} \mid \texttt{new C(}\overline{\texttt{e}}\texttt{)} & (\textit{expressions}) \\
 & & \mid \texttt{with L e} \mid \texttt{without L e} & \\
 & & \mid \texttt{proceed(}\overline{\texttt{e}}\texttt{)} \mid \texttt{super.m(}\overline{\texttt{e}}\texttt{)} & \\
 & & \mid \texttt{new C(}\overline{\texttt{v}}\texttt{)<C,}\overline{\texttt{L}}\texttt{,}\overline{\texttt{L}}\texttt{>.m(}\overline{\texttt{e}}\texttt{)} & \\
\texttt{v},\texttt{w} & ::= & \texttt{new C(}\overline{\texttt{v}}\texttt{)} & (\textit{values})
\end{array}$$

Following FJ, we use overlines to denote sequences: so, $\overline{\texttt{f}}$ stands for a possibly empty sequence $\texttt{f}_1, \cdots, \texttt{f}_n$ and similarly for $\overline{\texttt{C}}$, $\overline{\texttt{x}}$, $\overline{\texttt{e}}$, and so on. The empty sequence is denoted by •. We also abbreviate pairs of sequences, writing "$\overline{\texttt{C}}\ \overline{\texttt{f}}$" for "$\texttt{C}_1\ \texttt{f}_1, \cdots, \texttt{C}_n\ \texttt{f}_n$", where n is the length of $\overline{\texttt{C}}$ and $\overline{\texttt{f}}$, and similarly "$\overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{;}$" as shorthand for the sequence of declarations "$\texttt{C}_1\ \texttt{f}_1\texttt{;}\ldots\texttt{C}_n\ \texttt{f}_n\texttt{;}$" and "$\texttt{this.}\overline{\texttt{f}}\texttt{=}\overline{\texttt{f}}\texttt{;}$" for "$\texttt{this.f}_1\texttt{=f}_1\texttt{;}\ldots\texttt{;this.f}_n\texttt{=f}_n\texttt{;}$". We use commas and semicolons for concatenations. Sequences of field declarations, parameter names, layer names, and method declarations are assumed to contain no duplicate names.

A class definition CL consists of its name, its superclass name, field declarations $\overline{\texttt{C}}\ \overline{\texttt{f}}$, a constructor K, and method definitions $\overline{\texttt{M}}$. A constructor K is a trivial one that takes initial values of all fields and sets them to the corresponding fields. Unlike the examples in

[1] Variants of COP languages allow to manage layer compositions on a per-instance basis [9, 10], which is left as future work in the paper.

the last section, we do not provide syntax for layers; partial methods are registered in a partial method table, explained below. A method M takes $\overline{\mathtt{x}}$ as arguments and returns the value of expression e. As ContextFJ is a functional calculus like FJ, the method body consists of a single return statement and all constructs including with and without return values. An expression e can be a variable, field access, method invocation, object instantiation, layer activation/deactivation, proceed/super call, or a special expression $\mathtt{new\ C(\overline{v})<C,\overline{L},\overline{L}>.m(\overline{e})}$, which will be explained shortly. A value is an object of the form $\mathtt{new\ C(\overline{v})}$.

The expression $\mathtt{new\ C(\overline{v})<D,\overline{L}',\overline{L}>.m(\overline{e})}$, where $\overline{\mathtt{L}}'$ is assumed to be a prefix of $\overline{\mathtt{L}}$, is a special run-time expression and not supposed to appear in classes. It basically means that m is going to be invoked on $\mathtt{new\ C(\overline{v})}$. The annotation $\mathtt{<D,\overline{L}',\overline{L}>}$, which is used to model super and proceed, indicates where method lookup should start. More concretely, the triple $\mathtt{<D,(L_1;\cdots;L_i),(L_1;\cdots;L_n)>}$ $(i \leq n)$ means that the search for the method definition will start from class D of layer $\mathtt{L}_i$. So, for example, the usual method invocation $\mathtt{new\ C(\overline{v}).m(\overline{e})}$ (without annotation) is semantically equivalent to $\mathtt{new\ C(\overline{v})<C,\overline{L},\overline{L}>.m(\overline{e})}$, where $\overline{\mathtt{L}}$ is the active layers when this invocation is to be executed. This triple also plays the role of a "cursor" in the method lookup procedure and proceeds as follows

$$\begin{array}{ll}
 & \mathtt{<D,(L_1;\cdots;L_i),(L_1;\cdots;L_n)>} \\
\Rightarrow & \mathtt{<D,(L_1;\cdots;L_{i-1}),(L_1;\cdots;L_n)>} \quad \Rightarrow \quad \cdots \\
\Rightarrow & \mathtt{<D,\bullet,(L_1;\cdots;L_n)>} \\
\Rightarrow & \mathtt{<E,(L_1;\cdots;L_n),(L_1;\cdots;L_n)>} \quad \text{(E is a direct superclass of D)} \\
\Rightarrow & \mathtt{<E,(L_1;\cdots;L_{n-1}),(L_1;\cdots;L_n)>} \quad \Rightarrow \quad \cdots
\end{array}$$

until the method definition is found. Notice that the third element is needed when the method is not found in D in any layer including the base: the search continues to layer $\mathtt{L}_n$ of D's direct superclass.

With the help of this form, we can give a semantics of super and proceed by simple substitution-based reduction. For example, consider method invocation $\mathtt{new\ C().m(\overline{v})}$. As in FJ, this expression reduces to the method body where parameters and this are replaced with arguments $\overline{\mathtt{v}}$ and the receiver new C(), respectively. Now, what happens to super in the method body? It cannot be replaced with the receiver new C() since it would confuse this and super. Method lookup for super is different from usual (virtual) method lookup in that it has to start from the direct superclass of *the class in which* super *appears*. So, if the method body containing super.n() is found in class D, then the search for n has to start from the direct superclass of D. To express this fact, we replace super with new C()<E,...> where E is the direct superclass of D. We can deal with proceed similarly. Suppose the method body is found in layer $\mathtt{L}_i$ in D. Then, $\mathtt{proceed(\overline{e})}$ is replaced with $\mathtt{new\ C()<D,(L_1;\cdots;L_{i-1}),\overline{L}>.m(\overline{e})}$, where $\mathtt{L_1;\cdots;L_{i-1}}$ are layers activated before $\mathtt{L}_i$.

A ContextFJ program $(CT, PT, \mathtt{e})$ consists of a class table CT, which maps a class name to a class definition, a partial method table PT, which maps a triple C, L, and m of class, layer, and method names to a method definition, and an expression, which corresponds to the body of the main method. In what follows, we assume CT and PT to be fixed and satisfy the following sanity conditions:

1. $CT(\mathtt{C}) = \mathtt{class\ C\ \ldots}$ for any $\mathtt{C} \in dom(CT)$.
2. $\mathtt{Object} \notin dom(CT)$.
3. For every class name C (except Object) appearing anywhere in CT, we have $\mathtt{C} \in dom(CT)$;
4. There are no cycles in the transitive closure of the extends clauses.
5. $PT(\mathtt{m},\mathtt{C},\mathtt{L}) = \mathtt{\ldots\ m(\ldots)\{\ldots\}}$ for any $(\mathtt{m},\mathtt{C},\mathtt{L}) \in dom(PT)$.

$$\boxed{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}}$$

$$\mathit{fields}(\mathtt{Object}) = \bullet$$

$$\frac{\mathtt{class\ C \lhd D\ \{\ \overline{C}\ \overline{f};\ \ldots\ \}} \qquad \mathit{fields}(\mathtt{D}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}}}{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}}, \overline{\mathtt{C}}\ \overline{\mathtt{f}}}$$

$$\boxed{\mathit{mbody}(\mathtt{m},\mathtt{C},\overline{\mathtt{L}}',\overline{\mathtt{L}}) = \overline{\mathtt{x}}\mathtt{.e\ in\ D},\overline{\mathtt{L}}''}$$

$$\frac{\mathtt{class\ C \lhd D\ \{\ \ldots\ C_0\ m(\overline{C}\ \overline{x})\{\ return\ e;\ \}\ \ldots\}}}{\mathit{mbody}(\mathtt{m},\mathtt{C},\bullet,\overline{\mathtt{L}}) = \overline{\mathtt{x}}\mathtt{.e\ in\ C},\bullet}$$

$$\frac{PT(\mathtt{m},\mathtt{C},\mathtt{L}_0) = \mathtt{C\ m(\overline{C}\ \overline{x})\{\ return\ e;\ \}}}{\mathit{mbody}(\mathtt{m},\mathtt{C},(\overline{\mathtt{L}}';\mathtt{L}_0),\overline{\mathtt{L}}) = \overline{\mathtt{x}}\mathtt{.e\ in\ C},(\overline{\mathtt{L}}';\mathtt{L}_0)}$$

$$\frac{\begin{array}{c}\mathtt{class\ C \lhd D\ \{\ \ldots\ \overline{M}\ \}} \qquad \mathtt{m} \notin \overline{\mathtt{M}} \\ \mathit{mbody}(\mathtt{m},\mathtt{D},\overline{\mathtt{L}},\overline{\mathtt{L}}) = \overline{\mathtt{x}}\mathtt{.e\ in\ E},\overline{\mathtt{L}}'\end{array}}{\mathit{mbody}(\mathtt{m},\mathtt{C},\bullet,\overline{\mathtt{L}}) = \overline{\mathtt{x}}\mathtt{.e\ in\ E},\overline{\mathtt{L}}'}$$

$$\frac{PT(\mathtt{m},\mathtt{C},\mathtt{L}_0)\ \text{undefined} \qquad \mathit{mbody}(\mathtt{m},\mathtt{C},\overline{\mathtt{L}}',\overline{\mathtt{L}}) = \overline{\mathtt{x}}\mathtt{.e\ in\ D},\overline{\mathtt{L}}''}{\mathit{mbody}(\mathtt{m},\mathtt{C},(\overline{\mathtt{L}}';\mathtt{L}_0),\overline{\mathtt{L}}) = \overline{\mathtt{x}}\mathtt{.e\ in\ D},\overline{\mathtt{L}}''}$$

Figure 1. ContextFJ: Lookup functions.

Lookup functions. As in FJ, we define a few auxiliary functions to look up field and method definitions. They are defined by the rules in Figure 1. The function $\mathit{fields}(\mathtt{C})$ returns a sequence $\overline{\mathtt{C}}\ \overline{\mathtt{f}}$ of pairs of a field name and its type by collecting all field declarations from C and its superclasses. The function $\mathit{mbody}(\mathtt{m},\mathtt{C},\overline{\mathtt{L}}_1,\overline{\mathtt{L}}_2)$ returns the parameters and body $\overline{\mathtt{x}}\mathtt{.e}$ of method m in class C when the search starts from $\overline{\mathtt{L}}_1$; the other layer names $\overline{\mathtt{L}}_2$ keep track of the layers that are activated when the search initially started. It also returns the information on where the method has been found—the information will be used in reduction rules to deal with proceed and super. As we mentioned already, the method definition is searched for in class C in all activated layers and the base definition and, if there is none, then the search continues to C's superclass. By reading the rules in a bottom-up manner, we can read off the recursive search procedure. The first rule means that m is found in the base class definition C (notice the third argument is $\bullet$) and the second that m is found in layer $\mathtt{L}_0$. The third rule, which deals with the situation where m is not found in a base class (expressed by the condition $\mathtt{m} \notin \overline{\mathtt{M}}$), motivates the fourth argument of *mbody*. The search goes on to C's superclass D and has to take all activated layers into account; so, $\overline{\mathtt{L}}$ is copied to the third argument in the premise. The fourth rule means that, if C of $\mathtt{L}_0$ does not have m, then the search goes on to the next layer (in $\overline{\mathtt{L}}'$) leaving the class name unchanged.

3.2 Operational Semantics

The operational semantics of ContextFJ is given by a reduction relation of the form $\overline{\mathtt{L}} \vdash \mathtt{e} \longrightarrow \mathtt{e}'$, read "expression e reduces to $\mathtt{e}'$ under the activated layers $\overline{\mathtt{L}}$". Here, $\overline{\mathtt{L}}$ do not contain duplicate names, as we noted earlier. The main rules are shown in Figure 2.

The first four rules are the main computation rules for field access and method invocation. The first rule for field access is straightforward: *fields* tells which argument to new C(..) corresponds to $\mathtt{f}_i$. The next three rules are for method invocation. The second rule is for method invocation where the cursor of the method lookup procedure has not been "initialized"; the cursor is set to be at the receiver's class and the currently activated layers. In the third rule, the receiver is $\mathtt{new\ C(\overline{v})}$ and $\mathtt{<C',\overline{L}',\overline{L}>}$ is the location of the cursor. When the method body is found in the base-layer class

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}}{\overline{\mathtt{L}} \vdash \mathtt{new\ C(}\overline{\mathtt{v}}\mathtt{).f}_i \longrightarrow \mathtt{v}_i} \qquad \frac{\overline{\mathtt{L}} \vdash \mathtt{new\ C(}\overline{\mathtt{v}}\mathtt{)<C,}\overline{\mathtt{L}}\mathtt{,}\overline{\mathtt{L}}\mathtt{>.m(}\overline{\mathtt{w}}\mathtt{)} \longrightarrow \mathtt{e}}{\overline{\mathtt{L}} \vdash \mathtt{new\ C(}\overline{\mathtt{v}}\mathtt{).m(}\overline{\mathtt{w}}\mathtt{)} \longrightarrow \mathtt{e}}$$

$$\frac{\mathit{mbody}(\mathtt{m}, \mathtt{C}', \overline{\mathtt{L}}', \overline{\mathtt{L}}) = \overline{\mathtt{x}}\mathtt{.e}\ \mathtt{in}\ \mathtt{C}'', \bullet \qquad \mathtt{class\ C''}\triangleleft\ \mathtt{D\{...\}}}{\begin{array}{l}\overline{\mathtt{L}}''' \vdash \mathtt{new\ C(}\overline{\mathtt{v}}\mathtt{)<C',}\overline{\mathtt{L}}'\mathtt{,}\overline{\mathtt{L}}\mathtt{>.m(}\overline{\mathtt{w}}\mathtt{)} \longrightarrow \\ \qquad \left[\begin{array}{ll}\mathtt{new\ C(}\overline{\mathtt{v}}\mathtt{)} & /\mathtt{this}, \\ \overline{\mathtt{w}} & /\overline{\mathtt{x}}, \\ \mathtt{new\ C(}\overline{\mathtt{v}}\mathtt{)<D,}\overline{\mathtt{L}}\mathtt{,}\overline{\mathtt{L}}\mathtt{>} & /\mathtt{super}\end{array}\right]\mathtt{e}\end{array}}$$

$$\frac{\begin{array}{c}\mathit{mbody}(\mathtt{m}, \mathtt{C}', \overline{\mathtt{L}}', \overline{\mathtt{L}}) = \overline{\mathtt{x}}\mathtt{.e}\ \mathtt{in}\ \mathtt{C}'', (\overline{\mathtt{L}}''; \mathtt{L}_0) \\ \mathtt{class\ C''}\triangleleft\ \mathtt{D\{...\}}\end{array}}{\begin{array}{l}\overline{\mathtt{L}}''' \vdash \mathtt{new\ C(}\overline{\mathtt{v}}\mathtt{)<C',}\overline{\mathtt{L}}'\mathtt{,}\overline{\mathtt{L}}\mathtt{>.m(}\overline{\mathtt{w}}\mathtt{)} \longrightarrow \\ \qquad \left[\begin{array}{ll}\mathtt{new\ C(}\overline{\mathtt{v}}\mathtt{)} & /\mathtt{this}, \\ \overline{\mathtt{w}} & /\overline{\mathtt{x}}, \\ \mathtt{new\ C(}\overline{\mathtt{v}}\mathtt{)<C'',}\overline{\mathtt{L}}''\mathtt{,}\overline{\mathtt{L}}\mathtt{>.m} & /\mathtt{proceed}, \\ \mathtt{new\ C(}\overline{\mathtt{v}}\mathtt{)<D,}\overline{\mathtt{L}}\mathtt{,}\overline{\mathtt{L}}\mathtt{>} & /\mathtt{super}\end{array}\right]\mathtt{e}\end{array}}$$

$$\frac{\mathit{remove}(\mathtt{L}, \overline{\mathtt{L}}) = \overline{\mathtt{L}}' \qquad \overline{\mathtt{L}}'; \mathtt{L} \vdash \mathtt{e} \longrightarrow \mathtt{e}'}{\overline{\mathtt{L}} \vdash \mathtt{with\ L\ e} \longrightarrow \mathtt{with\ L\ e}'}$$

$$\frac{\mathit{remove}(\mathtt{L}, \overline{\mathtt{L}}) = \overline{\mathtt{L}}' \qquad \overline{\mathtt{L}}' \vdash \mathtt{e} \longrightarrow \mathtt{e}'}{\overline{\mathtt{L}} \vdash \mathtt{without\ L\ e} \longrightarrow \mathtt{without\ L\ e}'}$$

$$\frac{}{\overline{\mathtt{L}} \vdash \mathtt{with\ L\ v} \longrightarrow \mathtt{v}} \qquad \frac{}{\overline{\mathtt{L}} \vdash \mathtt{without\ L\ v} \longrightarrow \mathtt{v}}$$

Figure 2. ContextFJ: Reduction rules.

$\mathtt{C}''$ (denoted by "in $\mathtt{C}''$, $\bullet$"), the whole expression reduces to the method body where the formal parameters $\overline{\mathtt{x}}$ and this are replaced by the actual arguments $\overline{\mathtt{w}}$ and the receiver, respectively. Furthermore, super is replaced by the receiver with the cursor pointing to the superclass of $\mathtt{C}''$. The fourth rule, which is similar to the third, deals with the case where the method body is found in layer $\mathtt{L}_0$ in class $\mathtt{C}''$. In this case, proceed in the method body is replaced with the invocation of the same method, where the receiver's cursor points to the next layer $\overline{\mathtt{L}}''$ (dropping $\mathtt{L}_0$). Since the meaning of the annotated invocation is not affected by the layers in the context (note that $\overline{\mathtt{L}}'''$ are not significant in these rules), the substitution for super and proceed also means that their meaning is the same throughout a given method body, even when they appear inside with and without. Note that, unlike FJ, reduction in ContextFJ is call-by-value, requiring receivers and arguments to be values. This evaluation strategy reflects the fact that arguments should be evaluated under the caller-side context.

The following rules are related to context manipulation. The fifth rule means that e in with L e should be executing by activating L. The auxiliary function $\mathit{remove}(\mathtt{L}, \overline{\mathtt{L}})$, which removes L from $\overline{\mathtt{L}}$ (or returns $\overline{\mathtt{L}}$ if L is not in $\overline{\mathtt{L}}$), is used to avoid duplication of L. The next rule is similar: e is evaluated under the context where L is absent. The last two rules mean that, once the evaluation of the body of with/without is finished, it returns the value of the body.

There are other trivial congruence rules to allow subexpressions to reduce, but we omit them for brevity.

4. Type System

As usual, the role of a type system is to guarantee type soundness, namely, to prevent statically field-not-found and method-not-found errors from happening at run-time. In ContextFJ, it also means that a type system should ensure that every proceed() or super() call succeeds. However, it is not trivial to ensure this property, due to the dynamic nature of layer activation—the existence of a method definition in a given class may depend on whether a particular layer is activated.

Here, we give a simple type system, which is mostly a straightforward extension of FJ's type system but prohibits layers from introducing new methods that do not exist in the base-layer class—in other words, every partial method has to override a method of the same name in the base-layer class. As a result, the function *mtype* to retrieve a method type is the same as FJ's: it takes a method name and a class name as arguments and returns a pair, written $\overline{\mathtt{C}}\to\mathtt{C}_0$, of a sequence of the argument types $\overline{\mathtt{C}}$ and the return type $\mathtt{C}_0$. Its definition is given by the following rules.

$$\frac{\mathtt{class\ C}\ \triangleleft\ \mathtt{D\ \{...\ C_0\ m(}\overline{\mathtt{C}}\ \overline{\mathtt{x}}\mathtt{)\{\ return\ e;\ \}\ ...\}}}{\mathit{mtype}(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{C}}\to\mathtt{C}_0}$$

$$\frac{\mathtt{class\ C}\ \triangleleft\ \mathtt{D\ \{...}\ \overline{\mathtt{M}}\ \mathtt{\}} \qquad \mathtt{m} \not\in \overline{\mathtt{M}} \qquad \mathit{mtype}(\mathtt{m}, \mathtt{D}) = \overline{\mathtt{C}}\to\mathtt{C}_0}{\mathit{mtype}(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{C}}\to\mathtt{C}_0}$$

Subtyping. The subtyping relation C <: D is defined as the reflexive and transitive closure of the extends clauses.

$$\frac{}{\mathtt{C}\ \mathtt{<:}\ \mathtt{C}} \qquad \frac{\mathtt{C}\ \mathtt{<:}\ \mathtt{D} \qquad \mathtt{D}\ \mathtt{<:}\ \mathtt{E}}{\mathtt{C}\ \mathtt{<:}\ \mathtt{E}} \qquad \frac{\mathtt{class\ C}\ \triangleleft\ \mathtt{D\ \{...\}}}{\mathtt{C}\ \mathtt{<:}\ \mathtt{D}}$$

Typing. The type judgment for expressions is of the form $\mathcal{L}; \Gamma \vdash \mathtt{e} : \mathtt{D}$, read "e, which appears in $\mathcal{L}$, is given type D under Γ". Here, Γ denotes a type environment, which assigns types to variables—more formally, it is a finite mapping from variables to class names. $\mathcal{L}$, which stands for the location where e appears, is either $\bullet$, which means the top-level (i.e., under execution), C.m, which means method m in base class C, or L.C.m, which means m in class C in layer L. It is used in the typing rules for proceed() and super() calls. The type judgment for methods is of the form either M ok in C, read "method M is well-formed in base-layer class C", or M ok in L.C, read "partial method M is well-formed in layer L of class C." Finally, the type judgment for classes is of the form CL ok, read "class CL is well-formed." The typing rules are given in Figure 3.

The typing rules for expressions are straightforward. The first four rules for variables, field access, method invocation, and object instantiation are the same as those in FJ (except $\mathcal{L}$). The fifth and sixth rules for with and without, respectively, mean that a layer (de)activation is well typed if its body is well typed. The next rule means that $\mathtt{super.m}'(\overline{\mathtt{e}})$ has to appear in a method definition in some class C (not at the top level) and the type of $\mathtt{m}'$ is retrieved from C's superclass E. Otherwise, it is similar to the rule for method invocations. The rule for $\mathtt{proceed}(\overline{\mathtt{e}})$ is similar. The expression has to appear in a partial method definition, hence the location should be L.C.m. The final rule combines the rules for object instantiation and method invocation. Although the run-time type of the receiver is $\mathtt{C}_0$, the current cursor is at class D, which is a superclass of $\mathtt{C}_0$. So, the type of m is retrieved from D.

The typing rules for method definitions are straightforward also. Both rules check that the method body is well typed under the assumption that formal parameters $\overline{\mathtt{x}}$ are given declared types $\overline{\mathtt{C}}$ and this is given the name of the class name where the method appears. The type of the method body has to be a subtype of the declared return type. One notable difference in these rules is in the last premise. The first rule for base-layer methods means that the method may or may not be overriding; if it is overriding, the usual overriding condition is checked. Note that we allow covariant overriding of the return type. On the other hand, the second rule for a partial method means that it *has to override* the base-layer method *with exactly the same type*. We cannot allow covariant overriding because the order of layer composition vary at run-time.

A program $(\mathit{CT}, \mathit{PT}, \mathtt{e})$ is well-formed if $\mathit{CT}(\mathtt{C})$ ok for any $\mathtt{C} \in \mathit{dom}(\mathit{CT})$ and $\mathit{PT}(\mathtt{m}, \mathtt{C}, \mathtt{L})$ ok in L.C for any $(\mathtt{m}, \mathtt{C}, \mathtt{L}) \in$

Expression typing: $\boxed{\mathcal{L};\Gamma \vdash \texttt{e} : \texttt{C}}$

$$\frac{(\Gamma = \overline{\texttt{x}}\texttt{:}\overline{\texttt{C}})}{\mathcal{L};\Gamma \vdash \texttt{x}_i : \texttt{C}_i} \qquad \frac{\mathcal{L};\Gamma \vdash \texttt{e}_0 : \texttt{C}_0 \qquad \mathit{fields}(\texttt{C}_0) = \overline{\texttt{C}}\ \overline{\texttt{f}}}{\mathcal{L};\Gamma \vdash \texttt{e}_0\texttt{.f}_i : \texttt{C}_i}$$

$$\frac{\begin{array}{c}\mathcal{L};\Gamma \vdash \texttt{e}_0 : \texttt{C}_0 \qquad \mathit{mtype}(\texttt{m},\texttt{C}_0) = \overline{\texttt{D}} \rightarrow \texttt{D}_0 \\ \mathcal{L};\Gamma \vdash \overline{\texttt{e}} : \overline{\texttt{E}} \qquad \overline{\texttt{E}} \texttt{ <: } \overline{\texttt{D}}\end{array}}{\mathcal{L};\Gamma \vdash \texttt{e}_0\texttt{.m(}\overline{\texttt{e}}\texttt{)} : \texttt{D}_0}$$

$$\frac{\mathit{fields}(\texttt{C}_0) = \overline{\texttt{D}}\ \overline{\texttt{f}} \qquad \mathcal{L};\Gamma \vdash \overline{\texttt{e}} : \overline{\texttt{C}} \qquad \overline{\texttt{C}} \texttt{ <: } \overline{\texttt{D}}}{\mathcal{L};\Gamma \vdash \texttt{new C}_0\texttt{(}\overline{\texttt{e}}\texttt{)} : \texttt{C}_0}$$

$$\frac{\mathcal{L};\Gamma \vdash \texttt{e}_0 : \texttt{C}_0}{\mathcal{L};\Gamma \vdash \texttt{with L e}_0 : \texttt{C}_0} \qquad \frac{\mathcal{L};\Gamma \vdash \texttt{e}_0 : \texttt{C}_0}{\mathcal{L};\Gamma \vdash \texttt{without L e}_0 : \texttt{C}_0}$$

$$\frac{\begin{array}{c}\mathcal{L} = \texttt{C.m} \text{ or } \texttt{L.C.m} \qquad \texttt{class C} \lhd \texttt{E \{...\}} \\ \mathit{mtype}(\texttt{m}',\texttt{E}) = \overline{\texttt{D}} \rightarrow \texttt{D}_0 \qquad \mathcal{L};\Gamma \vdash \overline{\texttt{e}} : \overline{\texttt{E}} \qquad \overline{\texttt{E}} \texttt{ <: } \overline{\texttt{D}}\end{array}}{\mathcal{L};\Gamma \vdash \texttt{super.m}'\texttt{(}\overline{\texttt{e}}\texttt{)} : \texttt{D}_0}$$

$$\frac{\begin{array}{c}\mathcal{L} = \texttt{L.C.m} \\ \mathit{mtype}(\texttt{m},\texttt{C}) = \overline{\texttt{D}} \rightarrow \texttt{D}_0 \qquad \mathcal{L};\Gamma \vdash \overline{\texttt{e}} : \overline{\texttt{E}} \qquad \overline{\texttt{E}} \texttt{ <: } \overline{\texttt{D}}\end{array}}{\mathcal{L};\Gamma \vdash \texttt{proceed(}\overline{\texttt{e}}\texttt{)} : \texttt{D}_0}$$

$$\frac{\begin{array}{c}\mathit{fields}(\texttt{C}_0) = \overline{\texttt{D}}\ \overline{\texttt{f}} \qquad \mathcal{L};\Gamma \vdash \overline{\texttt{v}} : \overline{\texttt{C}} \qquad \overline{\texttt{C}} \texttt{ <: } \overline{\texttt{D}} \\ \texttt{C}_0 \texttt{ <: D} \qquad \mathit{mtype}(\texttt{m},\texttt{D}) = \overline{\texttt{F}}{\rightarrow}\texttt{F}_0 \qquad \mathcal{L};\Gamma \vdash \overline{\texttt{e}} : \overline{\texttt{F}} \qquad \overline{\texttt{E}} \texttt{ <: } \overline{\texttt{F}}\end{array}}{\mathcal{L};\Gamma \vdash \texttt{new C}_0\texttt{<D,}\overline{\texttt{L}}'\texttt{,}\overline{\texttt{L}}''\texttt{>(}\overline{\texttt{v}}\texttt{).m(}\overline{\texttt{e}}\texttt{)} : \texttt{F}_0}$$

Method/class typing: $\boxed{\texttt{M ok in C}}$ $\boxed{\texttt{M ok in L.C}}$ $\boxed{\texttt{CL ok}}$

$$\frac{\begin{array}{c}\texttt{C.m};\overline{\texttt{x}} : \overline{\texttt{C}},\texttt{this} : \texttt{C} \vdash \texttt{e}_0 : \texttt{D}_0 \\ \texttt{D}_0 \texttt{ <: C}_0 \qquad \texttt{class C} \lhd \texttt{D \{...\}} \\ \text{if } \mathit{mtype}(\texttt{m},\texttt{D}) = \overline{\texttt{E}} \rightarrow \texttt{E}_0\text{, then } \overline{\texttt{E}} = \overline{\texttt{C}} \text{ and } \texttt{C}_0 \texttt{ <: E}_0\end{array}}{\texttt{C}_0\texttt{ m(}\overline{\texttt{C}}\ \overline{\texttt{x}}\texttt{) \{ return e}_0\texttt{; \} ok in C}}$$

$$\frac{\begin{array}{c}\texttt{L.C.m};\overline{\texttt{x}} : \overline{\texttt{C}},\texttt{this} : \texttt{C} \vdash \texttt{e}_0 : \texttt{D}_0 \\ \texttt{D}_0 \texttt{ <: C}_0 \qquad \mathit{mtype}(\texttt{m},\texttt{C}) = \overline{\texttt{C}} \rightarrow \texttt{C}_0\end{array}}{\texttt{C}_0\texttt{ m(}\overline{\texttt{C}}\ \overline{\texttt{x}}\texttt{) \{ return e}_0\texttt{; \} ok in L.C}}$$

$$\frac{\begin{array}{c}\texttt{K} = \texttt{C(}\overline{\texttt{D}}\ \overline{\texttt{g}}\texttt{, }\overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{)\{ super(}\overline{\texttt{g}}\texttt{); this.}\overline{\texttt{f}}\texttt{=}\overline{\texttt{f}}\texttt{; \}} \\ \mathit{fields}(\texttt{D}) = \overline{\texttt{D}}\ \overline{\texttt{g}} \qquad \overline{\texttt{M}}\texttt{ ok in C}\end{array}}{\texttt{class C} \lhd \texttt{D \{ }\overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{ \} ok}}$$

Figure 3. ContextFJ: Typing rules.

dom(*PT*) and $\bullet;\emptyset \vdash \texttt{e} : \texttt{C}$ for some C, where $\emptyset$ is the empty type environment.

This type system is sound with respect to the operational semantics given in the last section:

THEOREM 1 (Subject Reduction). *Suppose given class and partial method tables are well-formed. If* $\bullet;\Gamma \vdash \texttt{e} : \texttt{C}$ *and* $\overline{\texttt{L}} \vdash \texttt{e} \longrightarrow \texttt{e}'$*, then* $\bullet;\Gamma \vdash \texttt{e}' : \texttt{D}$ *for some* D *such that* D <: C.

THEOREM 2 (Progress). *Suppose given class and partial method tables are well-formed. If* $\bullet;\emptyset \vdash \texttt{e} : \texttt{C}$*, then either* e *is a value or* $\overline{\texttt{L}} \vdash \texttt{e} \longrightarrow \texttt{e}'$ *for some* e′.

5. Discussion

Related Work The operational semantics of *cj*, a context-oriented extension to the *j* language family, is expressed using a delegation-based calculus [14]. Another approach to providing an operational semantics of COP layer constructs and their application is based on graph transformations [11]. Both approaches to representing context-dependent behavior *encode* COP programs into more general calculi. Our semantics, on the other hand, *directly* expresses context-dependent behavior.

Feature-oriented programming (FOP) [3] and delta-oriented programming (DOP) [12] also advocate the use of layers or delta modules respectively to describe behavioral variations. In both approaches, various similar software artifacts are obtained by *statically* composing layers with base-level classes. Thus, formal models of FOP [1, 5] and DOP [13] typically give translational semantics. Since they usually allow layers to add new methods, type systems that guarantee the translated program to be well typed with respect to the base language's type system are more sophisticated than ours.

Future Work The present type system may be too restrictive since it does not allow layers to introduce new methods. We are currently working on a more sophisticated type system that does not prevent method introduction by exploring some ideas from type systems for FOP.

References

[1] Sven Apel, Christian Kästner, and Christian Lengauer. Feature Featherweight Java: a calculus for feature-oriented programming and stepwise refinement. In *GPCE*, 2008. doi:10.1145/1449913.1449931.

[2] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, and Hidehiko Masuhara. ContextJ: Context-oriented programming with Java. *Computer Software*, 28(1):272–292, January 2011.

[3] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling stepwise refinement. *TSE*, 2004. doi:10.1109/TSE.2004.23.

[4] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming - an overview of ContextL. In *DLS*, 2005. doi:10.1145/1146841.1146842.

[5] Benjamin Delaware, William Cook, and Don Batory. A machine-checked model of safe composition. In *FOAL*, 2009. doi:10.1145/1509837.1509846.

[6] Robert Hirschfeld, Pascal Costanza, and Michael Haupt. An introduction to context-oriented programming with ContextS. In *GTTSE*, 2008.

[7] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *JOT*, 2008.

[8] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *TOPLAS*, 2001. doi:10.1145/503502.503505.

[9] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. EventCJ: A context-oriented programming language with declarative event-based context transition. In *Proc. of AOSD*, 2011. (to appear).

[10] Jens Lincke, Malte Appeltauer, Bastian Steinert, and Robert Hirschfeld. An open implementation for context-oriented layer composition in ContextJS. *SCP*, 2010. doi:10.1016/j.scico.2010.11.013.

[11] Tim Molderez, Hans Schippers, Dirk Janssens, Michael Haupt, and Robert Hirschfeld. A platform for experimenting with language constructs for modularizing crosscutting concerns. In *WASDeTT*, 2010.

[12] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In *SPLC*, 2010.

[13] Ina Schaefer, Lorenzo Bettini, and Ferruccio Damiani. Compositional type-checking for delta-oriented programming. In *AOSD*, 2011. (to appear).

[14] Hans Schippers, Dirk Janssens, Michael Haupt, and Robert Hirschfeld. Delegation-based semantics for modularizing crosscutting concerns. In *OOPSLA*, 2008. doi:10.1145/1449764.1449806.

Aspect Oriented Programming: a language for 2-categories

Nicolas Tabareau
INRIA
École des Mines de Nantes, France

ABSTRACT

Aspect-Oriented Programming (AOP) started ten years ago with the remark that modularization of so-called crosscutting functionalities is a fundamental problem for the engineering of large-scale applications. Originating at Xerox PARC, this observation has sparked the development of a new style of programming featured that is gradually gaining traction, as it is the case for the related concept of code injection, in the guise of frameworks such as Swing and Google Guice. However, AOP lacks theoretical foundations to clarify this new idea. This paper proposes to put a bridge between AOP and the notion of 2-category to enhance the conceptual understanding of AOP. Starting from the connection between the λ-calculus and the theory of categories, we propose to see an aspect as a morphism between morphisms—that is as a program that transforms the execution of a program. To make this connection precise, we develop an advised λ-calculus that provides an internal language for 2-categories and show how it can be used as a base for the definition of the weaving mechanism of a realistic functional AOP language, called MinAML.

Categories and Subject Descriptors

F.3.2 [**Semantics of Programming Languages**]: Algebraic approaches to semantics

General Terms

Languages, Theory, design

Keywords

AOP, 2-category

1. INTRODUCTION

Aspect-Oriented Programming. Aspect-Oriented Programming (AOP) [5] promotes better separation of concerns in software systems by introducing aspects for the modular implementation of crosscutting concerns. Indeed, AOP provides the facility to intercept the flow of control in an application and perform new computations. In this approach, computation at certain execution points, called *join points*, may be intercepted by a particular condition, called *pointcut*, and modified by a piece of code, called *advice*, which is triggered only when the runtime context at a join point meets the conditions specified by a pointcut. Using aspects, modularity and adaptability of software systems can be enhanced. In the AOP terminology, the algorithm that controls which aspects can be executed at each join point is called a *weaving* algorithm.

Much of the research on aspect-oriented programming has focused on applying aspects in various problem domains and on integration of aspects into full-scale programming languages such as Java. However, aspects are very powerful and the development of a weaving mechanism becomes rapidly a very complex task. While some research efforts [4, 15, 16] have made significant progress on understanding some of the semantic issues involved, the algebraic explanation of aspect features has never reached the beauty and simplicity of the connection between the λ-calculus and Cartesian closed categories. We believe that this is the main reason why AOP never found its place in theoretical computer science fields.

Giving a precise meaning to aspects in AOP is a fairly complicated task because the definition of a single piece of code can have a very rich interaction with the rest of the program, an interaction whose effect can come up at anytime during the execution. The main purpose of this paper is to formalize this interaction. Namely, we propose to put a bridge between AOP and the notion of 2-category. Starting from the connection between the λ-calculus and category theory, we propose to see an aspect as a 2-cell, that is as a morphism between morphisms. In the programming point of view, this means that an aspect can be seen as a program which transforms the execution of programs.

In this perspective, a weaving algorithm that defines the interaction of a collection of aspects with a given program will be understood as the computation of a normal form in the underlying 2-category of interest. Thus, an algorithm that is usually defined by hand and described coarsely in AOP systems becomes here a basic notion of rewriting theory. The definition of an internal language for Cartesian closed 2-category (2-CCC) will be the keystone of this paper, the basis to give a precise meaning to the possible interactions of a single aspect with the rest of the code.

λ-calculus and Cartesian closed categories. Category theory and programming languages are closely related. It is now folklore that the typed λ-calculus is the internal lan-

FOAL'11, March 21, 2011, Pernambuco, Brazil.

guage of Cartesian closed categories. Recall that Cartesian closed categories are categories equipped with a Cartesian product and such that each hom-set can naturally be seen as an object of the category (e.g. in the category **Set** of sets and functions, functions between two sets A and B form again a set). In this paradigm, objects of the category correspond to types in the typed λ-calculus and morphisms between objects A and B of the category correspond to λ-terms of type B with (exactly) one free variable of type A. The composition of morphisms corresponds to substitution, a notion that is at the heart of β-reduction—the fundamental rule of the λ-calculus.

This interpretation of the λ-calculus started in the early 80's from the work of John Lambek and Philip Scott [7, 8, 11]. Soon later, Robert Seely proposed a 2-categorical interpretation of the λ-calculus [12] where β-reduction constructs 2-cells between terms and their β-reduced version. This perspective is in line with the thought that 2-cells can be seen as rewriting rules between morphisms (or terms). This idea has been pushed further by Barnaby Hilken in [2] where he developed a 2-dimensional λ-calculus that corresponds to the free 2-category with lax exponentials.

Recall that a 2-category $\mathcal{C}$ is basically a category in which the class $\mathcal{C}(A, B)$ of morphisms between any objects A and B is itself a category. In other words, a 2-category is a category in which there exists morphisms

$$f : A \to B$$

between objects , and also morphisms

$$\alpha : f \Rightarrow g$$

between morphisms. The morphisms $f : A \to B$ are called *1-cells* and the morphisms $\alpha : f \Rightarrow g$ are called *2-cells*.

Seely's interpretation shows how typed λ-calculus can naturally be viewed as a 2-category. In this paper, we define an advised λ-calculus extending the typed λ-calculus with 2-dimensional primitives that enable to describe any 2-cell of a 2-CCC. Those additional primitives construct a kind of 2-dimensional terms that we will (by extension) call aspects. The resulting language, called λ_2-calculus, defines an internal language for Cartesian closed 2-category and will be the base of our explanation of aspects in AOP.

AOP and 2-categories. The keystone of this paper is to consider aspects in AOP as 2-cells in a 2-category just as functions (more precisely λ-terms) are interpreted as morphisms in a category. But this simple idea raises interesting and difficult issues:

- What is the good notion of variables at a 2-dimensional level?
- What is the extended notion of β-reduction?
- How to describe vertical and horizontal composition of a 2-category in the language of typed λ-calculi?

Once this effort to develop an internal language for Cartesian closed 2-categories has been done, it becomes simpler to describe the interaction of an aspect with the rest of a program. Indeed, the 2-dimensional constructors of the λ_2-calculus enable to faithfully describe all situations in which an aspect can be applied to a given program.

Let us anticipate on the description of the λ_2-calculus to give an example straightaway. Suppose that we have defined an aspect

$$\alpha : sqrt \Rightarrow sqrt \circ abs$$

which rewrites all calls to a square root function to ensure that inputs are non-negative. This aspect can be seen as a piece of advice whose pointcut intercepts the square root function and proceeds with the absolute value of the original argument of the function. The effect of α on the program

$$p = \lambda x.\ sqrt(sqrt(x))$$

will be described by the composed 2-cell

$$\beta = \lambda X.\ \alpha \circ \alpha \circ X : p \Rightarrow p'$$

that transforms the program p into the program

$$p' = \lambda x.\ sqrt(abs(sqrt(abs(x)))).$$

The aspect β is automatically generated from the aspect α and constructors of the λ_2-calculus. Note that one could argue that this violates one of the primary design goals of AOP, which is to allow separation of cross-cutting concerns. Indeed, each aspect is monomorphic in the sense that the aspect β in the example above is specific to the program p. It could seem unfortunate as an important goal of AOP languages is that the aspect may be oblivious to the target program – in 2-categorical/λ-calculus terms what seems needed is naturality/parametricity in the scaffolding. However, this is not the point of view adopted in the λ_2-calculus. The idea is that a single definition of the aspect α above will generate all the possible combinations of this aspect with 2-dimensional primitives of the language and other constant aspects.

Of course, existing AOP languages do not look like the λ_2-calculus so we show how programs of a simple functional language with aspects, introduced by David Walker and colleagues in [15] and called MinAML, can be translated into the λ_2-calculus. As claimed above, the semantics of such programs is provided by a *weaving* algorithm that corresponds to the computation of a normal form in the underlying 2-category.

At the end of this article, we explain how this algebraic account of AOP can drive the definition of aspects in more powerful languages extended with references, exceptions or any programming primitives that are well-understood in category theory. This could be done by using a 2-categorical version of computational monads introduced by Eugenio Moggi [10]—and used for example in Haskell—to extend the λ_2-calculus smoothly. This 2-categorical extension can be seen as a particular case of the recent work of Martin Hyland, Gordon Plotkin and John Power on enriched Lawvere theories [3].

Note that the work of Kovalyov [6] on modeling aspects by category theory is in accordance with the school of category theory for software design. In this paper, category theory is used as a foundational model for programming languages, which is a completely different line of work.

A full version of this article [13] is available.

2. THE λ_2-CALCULUS

2.1 Types, terms and aspects

The grammar of the λ_2-calculus generated by a set of sort names $\mathcal{S}$ is presented in Figure 1. The sets of types

$$\begin{array}{llll}
\text{types} & A & ::= & S \mid \mathbf{Unit} \mid A \times B \mid A \to B \\
\text{terms} & t & ::= & f \mid x \mid \mathbf{skip} \mid \lambda x.\ t \mid t(t) \mid \langle t, t\rangle \mid \pi_i(t) \\
\text{aspects} & \alpha & ::= & a \mid X \mid \ \texttt{asp.}\, t \mapsto t' \mid \alpha * \alpha \mid \alpha \circ \alpha \mid \langle \alpha, \alpha\rangle \mid \lambda X.\ \alpha
\end{array}$$

Figure 1: The grammar of λ_2-calculus

and terms is closed under the traditional λ-calculus operations. For the second dimension, we construct a set of aspects which transform terms into other terms. An aspect α that transforms the term t of type A into the term t' of type A will be noted

$$\alpha : (t \Rightarrow t') :: A$$

For every type A, we suppose given a denumerable set of variables $x, \ldots$ that induces a denumerable set of 2-variables

$$X : (x \Rightarrow x) :: A$$

All the constructions for pairing, abstraction and horizontal composition are extended to aspects and there is a notion of vertical composition $\alpha * \beta$ which means that the transformations performed by α and β are applied successively. We use the word "free" and "bound" in the usual sense for a 2-dimensional variable X in an aspect α. The main aspect forming operation

$$\texttt{asp.}\, t \mapsto t' : (t \Rightarrow t') :: A$$

defines an aspect that transforms a closed term t of type A into another closed term t' of type A. It is crucial in the construction that the two terms are closed. Indeed, we do not accept aspects of the form

$$\texttt{asp.}\, x \mapsto y : (x \Rightarrow y) :: A$$

where x and y are variables. Such an aspect would transform any term of type A into any term of type A.

We require that the class of aspects is closed under all aspect-forming operations except for the aspect $\texttt{asp.}\, t \mapsto t'$ which must always exist only for identity aspects on constant terms, that is when $t' = t$ is a constant term.

Note that there may be additional types (S), constant terms (f) and constant aspects (a) in the language. As for the `asp.` constructor, constant aspects must be defined on closed typed terms.

2.2 Typing rules

The typing rules of the λ_2-calculus are given in Figure 2. Terms are typed in the presence of a context Γ that stipulates the type of variables while aspects are typed in the presence of a context Δ that stipulates the type of 2-variables. The rules for terms are the standard rules for the λ-calculus.

Rule 2-Abstraction and Rule 2-Pairing are the higher order version of closure and product in the calculus. Rule 2-Application and Rule Vertical-composition are the reminiscence of the corresponding 2-categorical compositions. Observe that Rule Vertical-composition expects the same term t_2 at the common boundary of α and β.

Rule 2-Variable introduces a 2-variable in the context Δ. Rule Aspect checks that an aspect $\texttt{asp.}\, t \mapsto t'$ transforms a closed typed term t into another closed term t' of the same type.

Additional constant terms and aspects of the language are given with their specific typing rules. Equations between terms and between aspects, together with properties of the calculus are given in the long version [13].

3. CARTESIAN CLOSED 2-CATEGORIES AND THE λ_2-CALCULUS

The definition above leaves a lot of freedom. There are many λ_2-calculus. As it is the case for the traditional λ-calculus, one can think of *the* λ_2-calculus as the λ_2-calculus freely generated by a given set S of sort names, with no additional type, term, aspect or equation. But there are many more λ_2-calculi, as many as 2-categories as stated by the following proposition (details can be found in [13].

Proposition 1. *The λ_2-calculus is the internal language of Cartesian closed 2-categories.*

Using the correspondence between the λ_2-calculus and 2-CCCs, we can now define a weaving algorithm in terms of categorical rewriting.

As sketched in the introduction, given a term $t(x) : B$ of a λ_2-calculus $\mathcal{L}$ where x is of type A, we will consider all the possible interactions of the (constant) aspects defined in $\mathcal{L}$ with $t(x)$ by considering the category $\mathbf{C}(\mathcal{L})(A, B)$ of the 2-CCC $\mathbf{C}(\mathcal{L})$ associated to $\mathcal{L}$. This category contains all aspects that transform terms of type $A \to B$ and so the execution of an aspect corresponds to the application of a morphism in that category. Thus, the result of the weaving algorithm is given by the normal form of the image of $t(x)$ in that category.

$$\begin{array}{l}\text{Woven}(t(x)) = \{(t'(x), \alpha) \mid (x, t(x)) \xrightarrow{\alpha} (x, t'(x)) \\ \qquad\quad \text{is a maximal reduction in the category } \mathbf{C}(\mathcal{L})(A, B)\}\end{array}$$

Of course, such a normal form has no reason to be unique or even to exist.

Uniqueness of the normal form. Observe that all the work on *aspect composition* can be understood as a way to combine aspects while conserving uniqueness of the definition of the woven program. For example, when multiple pieces of advice can be applied at the same join point in AspectJ, precedence orders are (arbitrarily) defined, based on the order in which definitions of pieces of advice syntactically appear in the code. More algebraic approaches have been proposed (see eg. [9]).

Existence of a normal form. The absence of a normal form is often understood as a *circularity* in the application of aspects. This problem is difficult to overcome and can arise even in simple programs. For instance, the work of Eric Tanter on execution levels is precisely a way to introduce a hierarchy in the application of aspects and thus to avoid basic circular definition [14]. Other lines of work have proposed to restrict the power of pieces of advice (for example using a typing system [1]) in order to guarantee that the execution of the program is not critically perturbed.

Variable
$$\frac{}{\Gamma, x : A \vdash x : A}$$

Abstraction
$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.\ t : A \to B}$$

Application
$$\frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t(u) : B}$$

Bottom
$$\frac{}{\Gamma \vdash \textbf{skip} : \textbf{Unit}}$$

Pairing
$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash t' : B}{\Gamma \vdash \langle t, t' \rangle : A \times B}$$

Projection
$$\frac{}{\Gamma \vdash \pi_i^{A_1,A_2} : A_1 \times A_2 \to A_i}$$

2-Variable
$$\frac{}{\Delta, X : (x \Rightarrow x) :: A \vdash X : (x \Rightarrow x) :: A}$$

Aspect
$$\frac{\vdash t : A \qquad \vdash t' : A}{\Delta \vdash \texttt{asp}.\, t \mapsto t' : (t \Rightarrow t') :: A}$$

2-Pairing
$$\frac{\Delta \vdash \alpha : (t \Rightarrow t') :: A \qquad \Delta \vdash \beta : (u \Rightarrow u') :: B}{\Delta \vdash \langle \alpha, \beta \rangle : (\langle t, u \rangle \Rightarrow \langle t', u' \rangle) :: A \times B}$$

2-Abstraction
$$\frac{\Delta, X : (x \Rightarrow x) :: A \vdash \alpha : (t \Rightarrow t') :: B}{\Delta \vdash \lambda X.\ \alpha : (\lambda x.\ t \Rightarrow \lambda x.\ t') :: A \to B}$$

2-Application
$$\frac{\Delta \vdash \beta : (t \Rightarrow t') :: A \to B \qquad \Delta \vdash \alpha : (u \Rightarrow u') :: A}{\Delta \vdash \beta \circ \alpha : (t(u) \Rightarrow t'(u')) :: B}$$

Vertical-composition
$$\frac{\Delta \vdash \alpha : (t_1 \Rightarrow t_2) :: A \qquad \Delta \vdash \beta : (t_2 \Rightarrow t_3) :: A}{\Delta \vdash \alpha * \beta : (t_1 \Rightarrow t_3) :: A}$$

Figure 2: Typing rules of the λ_2-calculus

4. MinAML

This section gives the semantics of a concrete AOP language called MinAML by a translation to the λ_2-calculus. More precisely, given a program p, we will construct a λ_2-calculus λ_p, whose underlying 2-category defines a rewriting system from which we can deduce the definition of a weaving algorithm.

MinAML is a version (without conditionals and `before` and `after` advice) of the language introduced in [15] to give a first AOP language with a formal semantics. The absence of `before` and `after` advice is unimportant as they can both be encoded with an `around` advice.

4.1 Syntax

MinAML is an extension of the λ-calculus with products in two steps. The first extension is usual: we introduce declaration names that can be used to define names for terms of the language with the `let` constructor

$$\texttt{let}\, f = t.$$

We suppose given a set of declaration names, noted $f, g, \ldots$

The second extension is the introduction of aspects with the constructor

$$\texttt{around}\ f(x) = t$$

which indicates that at execution, the application of the function f with argument x is replaced by the term t. Using the terminology introduced at the beginning of the article, the term $f(x)$ defines the *pointcut* of the aspect and the term t defines its *advice*.

When declaring pieces of advice, the programmer can choose either to replace f entirely or to perform some computations interleaved with one (or more) execution of f (possibly with new arguments) using the keyword `proceed`.

In the same way, when multiple aspects intercept the same function f, one must define an order in the weaving mechanism. For simplicity, we have decided to choose the order of declaration in the program.

The grammar of MinAML is fully described in Figure 3. A program p is constituted of a list of declarations ds, a list of aspects α and a term t. The fact that there is only a global scope for aspects in our calculus is enforced by the stratified structure of a program. The term [ ] stands for the empty list, $[h]$ stands for the singleton list with element h and $l \cdot l'$ denotes the concatenation of lists.

Typing rules of MinAML are given in the long version of this article [13].

4.2 Extension to effectful aspects

So far, an aspect of MinAML is always pure. In order to exploit the full power of the λ_2-calculus, we need to add effectful aspects. We choose here to simply add two constants in the language: a logging function $log : \mathbb{N}at \to \mathbb{N}at$ whose purpose is to be intercepted by the logging aspect $\mathit{Log\_asp} : (log \Rightarrow \lambda x.\ x) :: \mathbb{N}at \to \mathbb{N}at.$ that transforms $log(n)$ into n and prints n to the screen.

4.3 A simple example

Let us now express in this language the example developed in the introduction. To make the example richer, we also define an aspect that applies the function log (before the function abs) to the argument of $sqrt$ so that the argument will then be printed out by the aspect $\mathit{Log\_asp}$. The following program of MinAML (where we use some usual primitives on integers) defines such aspects and run $sqrt$ on the negative value -4.

$$\begin{array}{rcl} \mathbb{P} & = & [\texttt{let}\, sqrt = \lambda x.\ \sqrt{x},\ \texttt{let}\, abs = \lambda x.\ |x|] \cdot \\ & & [\texttt{around}\ sqrt(x) = \texttt{proceed}(log(x)), \\ & & \mathit{Log\_asp}, \\ & & \texttt{around}\ sqrt(x) = \texttt{proceed}(abs(x))] \cdot \\ & & [sqrt(-4)] \end{array}$$

4.4 Weaving on a simple example

MinAML can be translated into the λ_2-calculus (see [13] for details) . Thus, the weaving algorithm can be deduced from the weaving algorithm described in Section 3.

In this short version, we will just explain its behavior on the simple program $\mathbb{P}$. The computation can be described by the following sequence of reductions (where some extra β-reduction has been performed to make the reading easier):

$$
\begin{array}{llll}
\text{types} & A & ::= & S \mid \mathbf{Unit} \mid A \times B \mid A \to B \\
\text{terms} & t & ::= & x \mid f \mid \mathbf{skip} \mid \lambda x.\ t \mid t(t) \mid \langle t, t \rangle \mid \pi_i(t) \mid \texttt{proceed}(t) \\
\text{aspects} & \alpha & ::= & [\,] \mid [\texttt{around}\ f(x) = t] \cdot \alpha \\
\text{declarations} & ds & ::= & [\,] \mid [\texttt{let}\ f = t] \cdot ds \\
\text{programs} & p & ::= & ds \cdot \alpha \cdot t
\end{array}
$$

Figure 3: The grammar of the MinAML

$$
\begin{array}{lll}
sqrt_1(-4) & \xrightarrow{a_1 \circ id(-4)} & sqrt_2(\log(-4)) \\
 & \xrightarrow{id(sqrt_2) \circ Log\_asp \circ id(-4)} & sqrt_2((\lambda x.\ x)(-4)) \\
 & & = sqrt_2(-4) \\
 & \xrightarrow{a_2 \circ id(-4)} & sqrt_3(abs_1(-4)) \\
 & \xrightarrow{a_3 \circ id(abs_1(-4))} & \sqrt{abs_1(-4)} \\
 & \xrightarrow{id(\sqrt{-}) \circ a_4 \circ id(-4)} & \sqrt{|-4|} = 2
\end{array}
$$

Observe the particular kind of *parametricity* describes in the introduction. Indeed, a single definition of the aspect *Log_asp* generates all the possible combinations of that aspect with 2-dimensional primitives of the language, and in particular the aspect

$$id(sqrt_2) \circ Log\_asp \circ id(-4)$$

used in the computation of the weaving for $sqrt_1(-4)$.

5. CONCLUSION

The idea of the paper is to approach AOP (and more generally type-preserving program transformation) from a category-theoretic perspective, in order to complement the software engineering approach. We believe that this approach could have substantial benefit at the level of conceptual understanding of what AOP actually is.

More precisely, we identify (Cartesian closed) 2-categories as a suitable setting in which programs can be seen as 1-cells and aspects (or more generally program transformations) can be seen as 2-cells. To make this analogy precise, we develop a language for 2-categories called the λ_2-calculus, as a 2-dimensional extension of the traditional λ-calculus, and show that it is an internal language for Cartesian closed 2-categories. We also show that the pure λ_2-calculus is strongly normalizing.

We then demonstrate the applicability of our construction by translating a more realistic functional AOP language called MinAML into the λ_2-calculus. This translation makes it possible to interpret a program of MinAML in a Cartesian closed 2-category and to define the weaving algorithm as the computation of a normal form in a rewriting system based on that 2-category. The well-foundedness of the weaving algorithm is thus given by the existence of a normal form in the corresponding rewriting system.

In the long version of this article, we discuss an algebraic way to extend the λ_2-calculus with various notions of computation using enriched Lawvere theory. This nice formulation of algebraic theories in an enriched setting enables to transpose the notion of computational monads of Eugenio Moggi at the level of 2-categories. We believe that this model-theoretic account of computation is necessary to understand the complex interaction between AOP mechanisms and traditional notions of computation.

6. REFERENCES

[1] D. Dantas and D. Walker. Harmless advice. In *8th*, volume 41, page 396, 2006.

[2] B. Hilken. Towards a proof theory of rewriting: the simply typed 2λ-calculus. *Theoretical Computer Science*, 170(1-2):407–444, 1996.

[3] M. Hyland, G. Plotkin, and J. Power. Combining effects: sum and tensor. *Theoretical Computer Science*, 357(1):70–99, 2006.

[4] R. Jagadeesan, A. Jeffrey, and J. Riely. A calculus of untyped aspect-oriented programs. In *Proceedings of ECOOP*, pages 54–73. Springer-Verlag, 2003.

[5] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of ECOOP*, volume 1241. Springer-Verlag, 1997.

[6] S. Kovalyov. Modeling Aspects by Category Theory. *FOAL 2010 Proceedings*, page 63, 2010.

[7] J. Lambek. Cartesian closed categories and typed lambda-calculi. In *13th Spring School on Combinators and Functional Programming Languages*, page 175. Springer-Verlag, 1985.

[8] J. Lambek and P. Scott. *Introduction to higher order categorical logic.* Cambridge University Press, 1988.

[9] R. Lopez-Herrejon, D. Batory, and C. Lengauer. A disciplined approach to aspect composition. In *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, page 77. ACM, 2006.

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

[11] P. Scott. Some aspects of categories in computer science. *Handbook of algebra*, 2:3–77, 2000.

[12] R. Seely. Modelling computations: a 2-categorical framework. In *2nd*, pages 65–71, 1987.

[13] N. Tabareau. Aspect oriented programming: a language for 2-categories (long version). Technical Report RR-7527, INRIA, 2011. http://hal.inria.fr/inria-00470400.

[14] É. Tanter. Execution levels for aspect-oriented programming. In *Proceedings of the 9th conference on AOSD*, pages 37–48, Rennes and Saint Malo, France, Mar. 2010. ACM Press.

[15] D. Walker, S. Zdancewic, and J. Ligatti. A theory of aspects. In *8th*, volume 38, pages 127–139, 2003.

[16] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transactions on Programming Languages and Systems*, 26(5):890–910, 2004.