Proving aspect-oriented programming laws

Leonardo Cole* lcn@cin.ufpe.br
Paulo Borba† phmb@cin.ufpe.br
Alexandre Mota acm@cin.ufpe.br

Informatics Center
Federal University of Pernambuco
P.O. Box 7851 - 50.732-970 Recife PE, Brazil

ABSTRACT

The proof of the behaviour-preserving property of programming laws is not trivially demonstrated. It is necessary to show that the programs, before and after the transformation, have the same behaviour. In this paper we show how it is possible to prove that an aspect-oriented programming law preserves behaviour; an operational semantics for Method Call Interception is used. An equivalence relation stating that two programs have the same behaviour is defined. We use these concepts and discuss soundness for the law *Add-Before Execution*.

Categories and Subject Descriptors

D.1 [**Software**]: Programming Techniques—*Aspect-Oriented Programming*; D.3.2 [**Programming Languages**]: Language Classifications—*AspectJ*

General Terms

Languages

Keywords

Refactoring, AspectJ, Aspect-Oriented Programming, Separation of concerns

1. INTRODUCTION

In order to explore the benefits of refactoring [7, 17, 18], aspect-oriented developers are identifying common transformations for aspect-oriented programs [16, 14, 10, 12], mostly in AspectJ [13], a general purpose aspect-oriented extension to Java [9]. However, they lack support for assuring that the transformations preserve behaviour and are indeed refactorings.

*Supported by CAPES.
†Partially supported by CNPq.

It is possible to use AspectJ programming laws [5] to derive or create behaviour preserving transformations (refactorings) for a subset of this language. Programming laws [11] define equivalence between two programs, given that some conditions are respected. By applying and composing those laws, one can show that an AspectJ transformation is a refactoring. A refactoring denotes a behaviour preserving transformation that increases code quality. Contrasting with a refactoring, a law is bi-directional and it does not always increase code quality, it is part of a bigger strategy that does. Besides, the laws are much simpler than most refactorings because they involve only localized changes, and each one focuses on one specific AspectJ construct. The laws form a basis for defining refactorings with confidence that they preserve behaviour. Hence, soundness of the laws with respect to a formal semantics is a necessary property.

This paper shows one way to prove that those aspect-oriented laws indeed preserve behaviour. We use the semantics of an aspect-oriented language [15] in which we can represent part of the laws. This language is not as expressive as AspectJ, but provides mechanisms to define some kinds of AspectJ advices with a well defined semantics. It allows us to explore notions of semantic equivalence between aspect-oriented programs. This increases the confidence that the transformations applied by the laws preserve behaviour. However, some hypothesis must be satisfied in order to enable the laws proof. For instance, the programs can not use reflection and can not be concurrent. Those hypothesis are also considered for object-oriented programming laws [4].

A limitation to our current work is a consequence of being able to represent only part of the laws with the chosen semantics. As the chosen language is not as powerful as AspectJ, we can represent Laws 3 - *Add before-execution*, 4 - *Add before-call*, 7 - *Add after-execution returning*, 13 - *Merge advices*, 15 - *Remove target parameter*, and 14 - *Remove this parameter*. It would be necessary to define another language (or extend the one we used) to prove the remaining laws. Nevertheless, we can use this subset of the laws to show that some important refactorings indeed preserve behaviour, for instance, the *Extract Method Calls* [14].

This paper is organized as follows. Section 2 discusses the semantics used here and our notion of equivalence between two programs. Section 3 introduces the laws, showing their structure, preconditions and intent. Section 4 shows a for-

mal argumentation about soundness of one law. Then, we discuss related work in Section 5 and conclude in Section 6.

2. SEMANTICS OF METHOD CALL INTERCEPTION (MCI)

Semantics for aspect-oriented languages is still en emerging field. The aspect-oriented languages used today still do not have an associated formal semantics where it is possible to reason about programs. However, there are several approaches [3, 1, 20, 15, 19, 6, 2] that try to solve this problem. In this section we discuss an aspect-oriented semantics based on Method Call Interception (MCI) [15].

The MCI semantics was chosen because it allows us to represent several of the advice types offered by AspectJ. Hence, allowing us to reason about programming laws involving those kinds of advice. Moreover, the MCI semantics is described as an extension to an object-oriented one, similarly to the way AspectJ extends Java. Therefore, providing an easier comprehension of how the semantics change from the object-oriented language to its aspect-oriented extension. This semantics only deals with advices, which we consider as a core concept in aspect-orientation. However, other AspetcJ constructs, such as inter-type declarations, are also important and the proof for laws involving them should consider a different or extended language.

Lämmel starts defining the semantics for a small java-like object-oriented language called μO^2 [15]. He describes an operational semantics and defines the rules for this language. Although Lämmel describes both static and dynamic semantics, we consider only the dynamic semantics because we want to compare behaviour of programs. The static semantics is useful to verify if the programs are well constructed according to the type system. Hence, the static semantics would be necessary to proof that the laws relate valid programs, this is regarded as a future work.

After defining the semantics for μO^2, Lämmel extends this language to incorporate the new construct `superimpose`, which allows the definition of an advice intercepting a method. However, the first definition for the `superimpose` construct is very simple and was extended in two ways. First he introduces interactivity, allowing advices to expose and use variables from the method's execution context. Second, he extends the language definition including quantitative mechanisms, allowing a single advice to intercept several methods. The syntax for the resulting aspect-oriented language can be seen in Figure 1. The MCI extension starts at the `caller` definition.

The `superimpose` construct defines that some code (*exp*) is to be executed on the occurrence of an event (*eve*). Comparing to AspectJ, the *exp* can be regarded as the advice body, and *eve* can be regarded as the pointcut expression. The description of an event defines when and where a method interception occurs. A method can be intercepted at three distinct points (*mci*): `dispatch`, before its arguments evaluation; `enter`, after the arguments evaluation but before the method's execution; and `exit`, after the method's execution. Those *mci* points are analogous to the `before-call`, `before-execution` and `after-returning-execution` from AspectJ. The other component of an event (*loc*) describes

prog	=	*cdef** *cn.mn*
cdef	=	`class` *cn* `extends` *cn* {*field** *mdef**}
field	=	*type fn*
mdef	=	*type mn* (*arg**) *body*
type	=	*cn* | `void`
arg	=	*type vn*
body	=	*exp* | `abstract`
cn	=	class names
fn	=	field names
mn	=	method names
vn	=	variable names
exp	=	`null`
	|	`this`
	|	*vn*
	|	`view` *type exp*
	|	*exp.fn*
	|	*exp.vn* = *exp*
	|	*exp.mn* (*exp**)
	|	`super.`*mn* (*exp**)
	|	`let` *vn* : *type* = *exp* `in` *exp*
	|	*exp;exp*
	|	`while` (*exp*) *exp*
	|	`caller`
	|	`callee`
	|	`superimpose` *exp* `on` *eve*
eve	=	*mci loc* | *eve* `within` *loc*
mci	=	`dispatch` | `enter` | `exit`
loc	=	*
	|	`object` *exp*
	|	`class` *cn*
	|	`subclass` *cn*
	|	`method` *mn*
	|	`result` *type*
	|	`argument` *type vn*
	|	*loc* && *loc*
	|	*loc* || *loc*
	|	!*loc*

Figure 1: MCI syntax

the location of the method interception, which is an expression that matches methods based on its name, class, arguments, return type, etc. An event can also be constrained to occur only within another location.

Lämmel defines an operational semantics for this language [15]. He defines several rules to show how an expression should be evaluated. Each rule shows the return value of the evaluated expression and shows how the state changes. Some rules may depend on the the execution of other rules to achieve its result. Hence, the evaluation of a program can be represented as a tree showing several evaluation rules.

The domains for a rule consist of a method code table (T), which links method names with its parameters and body. An object store (Σ) that holds references to objects and its field values. This object store also hold the advice registry. There is also a reference to the executing object (θ) and an environment for the program variables (η). The expression $\Pi_i(t)$ denotes the *ith* projection of a tuple t.

Figure 2 shows the evaluation rule [15] for the `superimpose` construct. This rule states that evaluating a `superimpose` declaration returns a `null` reference (0 is the meaning of a null expression) and updates the object store (Σ''). The superimpose evaluation consists of three steps: first, we eval-

uate the event expression (1), which yields the event description ($\overline{k}$) and an updated object store (Σ'); second, we create the advice, represented by α (2); finally, we call the `register` helper function (3), which updates Σ' by registering the event and advice from the previous evaluations.

$$
\begin{array}{ll}
T, \Sigma, \theta, \eta \vdash eve \Rightarrow \overline{k}, \Sigma' & (1)\\
\wedge\ \alpha = ((\Pi_{cn}(\theta), \Pi_{mn}(\theta)), exp) & (2)\\
\wedge\ \overline{\texttt{register}}(\Sigma', \overline{k}, \alpha) \Rightarrow \Sigma'' & (3)\\
\hline
T, \Sigma, \theta, \eta \vdash \texttt{superimpose}\ exp\ \texttt{on}\ eve\ \Rightarrow 0, \Sigma'' & (4)
\end{array}
$$

Figure 2: `superimpose` evaluation rule

We do not show all the evaluation rules, more details can be found elsewhere [15]. As mentioned before, one of the reasons to choose the MCI semantics is that it shows an object-oriented semantics and extends it to introduce MCI. This description allows us to see exactly how the semantics change when we introduce aspect-oriented features to the language. As the superimpose construct affects only method calls, the only rule changed during the MCI extension is the method `call` evaluation rule.

Originally a method call is evaluated according to the rule listed in Figure 3. First, we evaluate the expression that yields the object on which the method is being called (5). Second, we search the environment for the method definition (6). Then, it is necessary to evaluate the expressions representing the arguments values (7-9). Finally, an environment is mounted with the evaluated arguments (10) to execute the method's body (11).

$$
\begin{array}{ll}
T, \Sigma_0, \theta, \eta \vdash exp \Rightarrow \rho, \Sigma_1 & (5)\\
\wedge\ \Pi_1(T) \bullet (\rho, mn) = ((vn_1, ..., vn_n), exp') & (6)\\
\wedge\ T, \Sigma_1, \theta, \eta \vdash exp_1 \Rightarrow v_1, \Sigma_2 & (7)\\
\wedge\ ... & (8)\\
\wedge\ T, \Sigma_n, \theta, \eta \vdash exp_n \Rightarrow v_n, \Sigma_{n+1} & (9)\\
\wedge\ \eta' =\!\perp [vn_1 \mapsto v_1, ..., vn_n \mapsto v_n] & (10)\\
\wedge\ T, \Sigma_{n+1}, \eta' \vdash exp' \Rightarrow v, \Sigma_{n+2} & (11)\\
\hline
T, \Sigma_0, \theta, \eta \vdash exp.mn(exp_1, ..., exp_n) \Rightarrow v, \Sigma_{n+2} & (12)
\end{array}
$$

Figure 3: Object-oriented `call` evaluation rule

A general object reference is represented by ρ. The function application is denoted as $f \bullet x$, and the entirely undefined function is denoted as $\perp$. The evaluation of a method call yields its value (v') and an updated object store (Σ'_{n+2}).

With the MCI extension, the `call` rule is changed to verify at certain points, if there is a registered event that should be executed. Figure 4 shows the `call` rule with the MCI extension. The lookup for registered events matching this method's execution is done through the helper functions `dispatch` (15), `enter` (20), and `exit` (22).

An event can be registered using the `superimpose` construct. The lookup functions showed in the MCI call rule, search the

$$
\begin{array}{ll}
T, \Sigma_0, \theta, \eta \vdash exp \Rightarrow \rho, \Sigma_1 & (13)\\
\wedge\ \Pi_1(T) \bullet (\rho, mn) = ((vn_1, ..., vn_n), exp') & (14)\\
\wedge\ \texttt{dispatch}(T, \Sigma_1, \theta, (\rho, mn)) \Rightarrow \Sigma'_1 & (15)\\
\wedge\ T, \Sigma'_1, \theta, \eta \vdash exp_1 \Rightarrow v_1, \Sigma_2 & (16)\\
\wedge\ ... & (17)\\
\wedge\ T, \Sigma_n, \theta, \eta \vdash exp_n \Rightarrow v_n, \Sigma_{n+1} & (18)\\
\wedge\ \eta' =\!\perp [vn_1 \mapsto v_1, ..., vn_n \mapsto v_n] & (19)\\
\wedge\ \texttt{enter}(T, \Sigma_{n+1}, \theta, (\rho, mn), \eta') \Rightarrow \Sigma'_{n+1} & (20)\\
\wedge\ T, \Sigma'_{n+1}, ((\rho, mn), \perp) \vdash exp' \Rightarrow v, \Sigma_{n+2} & (21)\\
\wedge\ \texttt{exit}(T, \Sigma_{n+2}, \theta, (\rho, mn), \eta', v) \Rightarrow v', \Sigma'_{n+2} & (22)\\
\hline
T, \Sigma_0, \theta, \eta \vdash exp.mn(exp_1, ..., exp_n) \Rightarrow v', \Sigma'_{n+2} & (23)
\end{array}
$$

Figure 4: MCI `call` evaluation rule

environment to see if the registered event matches the executing method. If there is a match, the registered expression is executed. Note that the `superimpose` must be evaluated before the method call for the advice to take effect. Any method calls made before the `superimpose` evaluation will behave according to the μO^2 rule because the environment will not have a registered event. This feature allows us to dynamically introduce advices, which is not possible in AspectJ.

As we want to map the MCI semantics to AspectJ, we need to constrain the language to ensure that all `superimpose` expressions are evaluated before the program starts executing. This can be achieved by allowing `superimpose` declarations only at the beginning of the main method (method called to initiate the program execution according to the language grammar, see *prog* in Figure 1).

It is possible to represent part of the advice types provided by AspectJ using the `superimpose` construct. In fact, we can represent `before-call`, `before-execution` and `after-returning-execution` advices. The first type maps to a `superimpose on dispatch` construct, the other two can be mapped to `superimpose on enter` and `superimpose on exit` constructions, respectively.

Other AspectJ constructs, including pointcuts, inter-type declarations, and other kinds of advice, can not be represented with the MCI semantics. This limitation enables us to reason only about Laws 3 - *Add before-execution*, 4 - *Add before-call*, 7 - *Add after-execution returning*, 13 - *Merge advices*, 15 - *Remove target parameter*, and 14 - *Remove this parameter*. In Section 4 we discuss the soundness of Law 3 - (*Add Before-Execution*). To enable the proof of the other laws, it would be necessary to extend the showed language, or to define a completely new one. This is regarded as a future work.

2.1 MCI Program Equivalence

We want to use the MCI semantics to reason about aspect-oriented programs and verify whether two programs behave the same. Thus, it is necessary to define an equivalence relation between them. This equivalence relation can be

difficult to define. For instance, if we choose an equivalence relation that compares two environments (states) resulting from programs execution, it would fail to compare programs that behave the same but use different data structures. Different data structures may result in different environments at the end of a program execution. For example, consider two stack implementations: the first uses an array to represent the stack, and the second uses a linked list. Both implementations may behave as a stack, but their final states are different because their data structures are different. In this case, it would be necessary to isolate input and output variables from the environment and compare only those variables.

As the programming laws we are willing to proof, with the MCI semantics, do not change the data structure, we can establish equivalence by comparing the object stores generated by the evaluation of both programs. Figure 5 shows the object store (Σ) domain to evaluate an expression [15]. This domain has three components: a function that associates data locations with their values ($\delta \rightarrow_{fin} v$), a function that associates object references with their types ($\rho \rightarrow_{fin} cn$), and the advice registry ($\overline{\Delta}$). Our equivalence notion only uses the first component of the object store comparing the field values and how they change, as stated by Definition 1.

Σ	$=$	$\delta \rightarrow_{fin} v$	(Object store)
	$\times$	$\rho \rightarrow_{fin} cn$	(Runtime type information)
	$\times$	$\overline{\Delta}$	(Advice registry)
δ	$=$	$\rho \times fn$	(Data locations)
ρ			(Object references)

Figure 5: Object Store

Definition 1 (Program Equivalence). *Let P and Q be two MCI programs. P is equivalent to Q ($P \equiv Q$) iff, for all valid input, the fields and their values from the resulting object store of P equals that of Q.*

We are only interested in the first component from the object store, which maps field locations to their values. Thus, after the programs evaluation we can compare the values of their fields and state that two programs behave the same if all their fields and values are equal. The runtime type information is not relevant to our relation, it is part of the object store to allow the evaluation of expressions like type casts. The advice registry is expected to change because we intend to introduce new `superimpose` commands to the program.

This equivalence notion is rather strong. It may distinguish two programs even if they have the same behaviour. The stack implementations using an array or a linked list would be different programs according to our definition. This is not a problem because two programs that are equivalent according to our definition, would also be equivalent using a more precise definition. Besides, our definition is the simplest solution suitable to our goals. Also, note that we are interested on the external behaviour of a program. Hence, our definition deals with closed programs and not with equivalence of classes. For instance, a method never called by a program do not influence the equivalence notion because its behaviour do not contribute to the external program behaviour.

Although we define the equivalence relation for MCI, this notion is independent of programming languages. However, this equivalence relation can only be considered for sequential programs. If the programs are concurrent, the equivalence relation should consider the structure of the evaluation tree as well. Nevertheless, our laws do not deal with those mechanisms.

3. LAWS

Sometimes, modifications required by refactorings are difficult to understand as they might perform global code changes. We use programming laws [5] to increase the confidence that an AspectJ transformation preserves behaviour. The laws are much simpler than most refactorings because they involve only localized changes, and each one focuses on one specific AspectJ construct. The laws form a basis for defining refactorings with confidence that they preserve behaviour.

In this section we describe a simple law, showing its intent, structure, and preconditions. The laws establish the equivalence of AspectJ programs given that some restrictions are respected. Therefore, the structure of each law consists of three parts: left-side, right-side and preconditions. The first two are templates of the equivalent programs. The third part indicates conditions that must hold to ensure the equivalence is valid. For example, the following law is useful to extract code from the beginning of a method into an aspect. If the extracted code is spread through several methods, we would apply the law several times to isolate this code. Afterwards, we would use another law to merge the resulting advices, increasing reuse.

Law 3. Add Before-Execution

```
ts                                  ts
class C {                           class C {
  fs                                  fs
  ms                                  ms
  T m(ps) {                           T m(ps) {
    body′;                              body
    body                            }
  }                                 }
}                                   paspect A {
paspect A {                           pcs
  pcs                                 bars
  bars                                before(context) :
  afs                                  exec(σ(C.m)) &&
}                                      bind(context) {
                                       body′[cthis/this]
                                    }
                                    afs
                                  }
```

provided

- ($\rightarrow$) *body′* does not declare or use local variables; *body′* does not call `super`;
- ($\leftarrow$) *body′* does not call `return`;
- ($\leftrightarrow$) *A* has the lowest precedence on the join points involving the signature $\sigma(C.m)$;

The laws basically represent two transformations, one applying the law from left to right and another one in the opposite direction. Each law has preconditions to ensure that the program is valid after the transformation and preconditions to ensure that the transformation preserves behaviour. When applied from left to right, this law moves part of a method's body into an advice that is triggered before method execution.

We denote the set of class and aspect declarations by *ts*, and the set of field declarations and method declarations by *fs* and *ms*, respectively. We also abstract the `privileged` modifier from AspectJ as `priv`. The set of pointcut declarations is denoted as *pcs*. Note that the advices can not be considered as a set, since order of declaration dictates precedence of advices. According to the AspectJ semantics, if two advices are *after*, the one declared later has precedence, in every other case, the advice declared first has precedence. Thus, we divide the list of advices in two. The first part (*bars*) contains the list of all `before` and `around` advices, while the second part contains only `after` advices (*afs*). This separation ensures that `after` advices always appear at the end of the aspect. It also allows us to define exactly the point where the new advice should be placed to execute in the same order in both sides of the law. Additionally, for advices declared in different aspects, precedence depends on their hierarchy or their order in a `declare precedence` construct.

Inside advices, we can access variables in the context of the captured join point. The law always expose the maximum context available, in this case, the executing object (`this`(*cthis*)) and the method parameters (`args`(*ps*)). The expression *bind(context)* includes those pointcut designators for exposing context. We omit visibility modifiers, `throws` clauses and inheritance constructs for simplicity. However, there are similar laws that include the variations of visibility modifiers, exceptions and inheritance constructs.

Examining the left hand side of Law 3, we see that $body'$ executes after all `before` advices declared for this join point. It also executes after all the `around` advices, intercepting this join point, call `proceed`. This means that the new advice on the right hand side of the law should be the last one to execute, preserving the order in which the code is executed in both sides of the law. Thus, the `before` advice should be placed after the list of `before` and `around` advices, but before the list of `after` advices. Moreover, to ensure that the new advice created with Law 3 is the last one to execute, we have a precondition stating that aspect A has the lowest precedence over other aspects defined in *ts*. This precondition must hold in both directions.

As we move $body'$ to the aspect, its visible context changes. Hence, it is necessary to constrain the context dependencies in order to guarantee that the law relates valid AspectJ programs. Therefore, we impose conditions on accessing private members, local variables (not including the methods arguments) and calls to `super`. While the last two are forbidden, access to private members is allowed if done through `this`. This is necessary to enable the mapping of accesses to the object referenced by `this`, to the object exposed as the executing object on the advice (*cthis*). The mapping is denoted by the expression $body'[cthis/\texttt{this}]$, where we substitute all occurrences of `this` for the variable *cthis* in $body'$.

However, there are other implications that must be considered. Changes to the method execution flow (calls to `return`) are generally not allowed because the advice cannot implement it, or it would increase complexity. This precondition is necessary to ensure that the law preserves behaviour.

Other laws are similarly defined in terms of transformations and preconditions, and establish properties of other constructs besides `before` advice. Table 1 shows a summary of the laws. More details about AspectJ programming laws can be found elsewhere [5].

4. SOUNDNESS OF THE ADD BEFORE-EXECUTION LAW

In this section we show that the Law 3 (*Add Before-Execution*) is sound using the semantics we chose. We interpret both sides of the law according to the semantics. Then we compare the resulting environments according to our equivalence notion to see whether the two sides of the law have the same meaning.

Following, we show the Law 3 written in terms of the MCI syntax. Thus, we map the `before-execution` advice from AspectJ to a `superimpose on enter` construct from the MCI language (see Section 2). Also, we constrain the language allowing only declarations of the `superimpose` construct at the beginning of the main method. Moreover, the MCI language does not have any modular concept similar to an aspect. Thus, the aspect simulation is also accomplished by the use of a main method with `superimpose` declarations at the beginning. As a consequence, changes made to the aspect are represented as changes made to the main method and its superimposes. Note that, similarly to the AspectJ law, we have to substitute the `this` keyword for the `callee` keyword when using $body'$ on the right hand side of the law.

Law 3. Add Before-Execution (MCI)

```
ts
class C ext T {
 fs
 ms
 Type m(ps) {
   body';
   body
 }
}
class M ext T {
 void main() {
   sis;
   mainBody
 }
}
```

=

```
ts
class C ext T {
 fs
 ms
 Type m(ps) {
   body
 }
}
class M ext T {
 void main() {
   superimpose body'
     on enter
       class C &&
       method m &&
       argument ps;
   sis;
   mainBody
 }
}
```

Table 1: Summary of laws

Law	Name	Law	Name
1	Add empty aspect	16	Remove argument parameter
2	Make aspect privileged	17	Add catch softened exception
3	Add before-execution	18	Soften exception
4	Add before-call	19	Remove exception from throws clause
5	Add after-execution	20	Remove exception handling
6	Add after-call	21	Move exception handling to aspect
7	Add after-execution returning successfully	22	Move field to aspect
8	Add after-call returning successfully	23	Move method to aspect
9	Add after-execution throwing exceptions	24	Move implements declaration to aspect
10	Add after-call throwing exceptions	25	Move extends declaration to aspect
11	Add around-execution	26	Extract named pointcut
12	Add around-call	27	Use named pointcut
13	Merge advices	28	Move field introduction up to interface
14	Remove `this` parameter	29	Move method introduction up to interface
15	Remove `target` parameter	30	Remove method implementation

There is also the advice ordering problem discussed in Section 3. According to our understanding from the MCI semantics, advices declared later have precedence, no matter the kind of MCI. Thus, we do not need to separate advices as we do with AspectJ. It is only necessary to declare the new `superimpose on enter`, just before all the other `superimpose` declarations (*sis*) to ensure that the new one is the last to be executed. If we were dealing with Law 7 (*Add after-execution returning successfully*), the new `superimpose` declaration should be placed after all the existing ones to ensure that the `after` advice should be the first to execute. We assume that the kind of rewriting discussed so far, does not change the semantics of Law 3.

In Section 2 we showed that there is just one evaluation rule that changes with the MCI extension. Thus, our soundness discussion involves only the `call` rule. A complete proof would involve all the language constructs and use induction on the structure of *mainBody*. The base case would consider each single command that can appear in *mainBody*, while the induction step would consider every composition of those commands. This complete proof is regarded as a future work, here we provide a formal argumentation to show that Law 3 is sound.

Our argumentation is based on a case where the *mainBody* represents a single call to method m of class C (note that we need to create an object, using the `let` construct, to call a method). This comes directly from the fact that the `superimpose` only affects the method call semantics. Any other simple construction for *mainBody* would trivially preserve behaviour because the other language constructs are not affected by the `superimpose`.

Figure 6 shows the evaluation tree for the left hand side of the law, considering that *mainBody* is the command: `let` c : C = `new` C `in` $c.m(ps)$. Every node consists of a program state. The transitions represent applications of transition rules according to the semantics. Thus, each transition is labeled after the applied rule. Also, the left square represent the input object store and the right square represents the output object store for each rule applied. The nodes are numbered according to the execution order, with label L1 being the first.

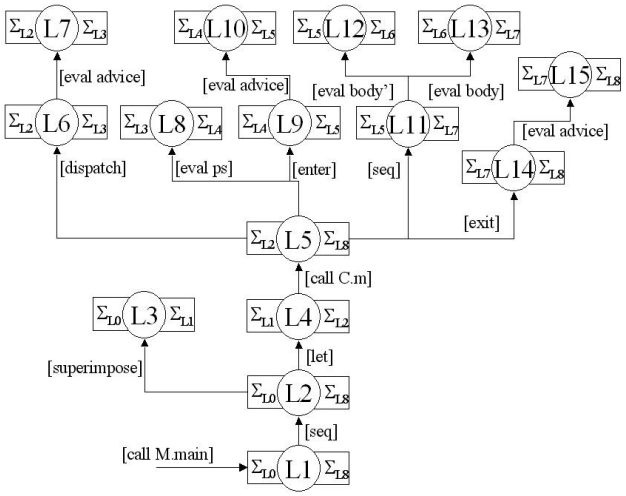

Figure 6: Evaluation tree for the left hand side.

The left hand side consists in evaluating a sequential composition (L2), which leads to the evaluation of the `superimpose` declarations present in *sis* (L3) and the evaluation of the `let` command (L4). The let updates the store and calls method m of class C (L5). The method call evaluation occurs as showed in Figure 4. First, events registered for `dispatch` MCI are executed (L7). Next we evaluate the method's parameters (L8). Then, events registered for `enter` MCI are executed (L10). Following we evaluate the method's body, which is a sequential composition (L11) of body' (L12) and body (L13). Finally, events registered for `exit` MCI are executed (L15). As we want to compare the execution of two programs, we do not expand execution nodes that are equal for both. For instance, the evaluation of *body*, *body'*, *ps*, `dispatch` and `exit` advice nodes are the same for both programs.

Next, Figure 7 shows the evaluation tree for the right hand side of the law. In this case, there is a sequential composition(R2) that first evaluates another sequential composition (R3), which includes our new `superimpose` (R4) and the old ones (R5). Then it starts the program similarly to the left hand side. The evaluation of the `superimpose` command updates the registry located on the object store by regis-

tering *body′* to be executed when entering the method *m* with arguments *ps* of class *C*. As a result, the evaluation of the `enter` helper function (R11) performs a lookup in the registry for events registered for this method and finds that *body′* should be executed (R12). Another difference is that the evaluation of the method's body now includes only *body* (R14).

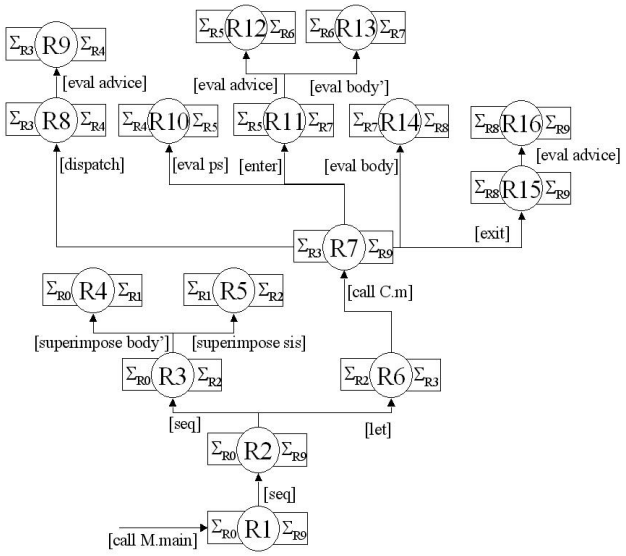

Figure 7: Evaluation tree for the right hand side.

Proof. (Sketch) According to the equivalence notion established in Section 2.1, we are interested on the nodes that may update the first component of the object store (field values). First, the `let` command may update the object store by adding a new object and the values of its fields. The second way to update the field values in the object store is through an assignment. Assignments can appear in any expression and thus, we look for the nodes able to evaluate expressions.

On the left hand side, the nodes related to the evaluation of expressions are: `let` (L4), `dispatch` (L7), *ps* (L8), `enter` (L10), *body′* (L12), *body* (L13), and `exit` (L15). Similarly, the nodes we are interested on the right hand side are: `let` (R6), `dispatch` (R9), *ps* (R10), `enter` (R12), *body′* (R13), *body* (R14), and `exit` (R16).

Analyzing the equivalent nodes from both programs (i.e. L4 and R6, L7 and R9, etc) we can see they are syntactically equal, and thus have an equivalent evaluation. The only factor that may result in different field values at the end of the program execution is the order in which the nodes are evaluated. In both Figures 6 and 7, the number inside the node represent the order of evaluation, which is the same in both programs. During the evaluation, the field values are supposed to be equal after the evaluation of nodes L13 and R14. Thus, according to our equivalence notion, and considering that the programs are sequential, we can conclude that the programs have the same behaviour. ■

4.1 Soundness of Other Laws

This proof could be similarly extended for Laws 4 (*Add before-call*), and 7 (*Add after returning successfully*). As they only differ by the kind of advice (MCI) used. Law 4 would use the `superimpose on dispatch` construct and Law 7 would use the `superimpose on exit` construct. For this reason, we consider that this two laws are also sound.

The right hand side of Law 4 would generate an evaluation tree where *body′* is evaluated before some other method call. This means that *body′* is evaluated even before the arguments of the method to be called. The evaluation tree for the left hand side would place *body′* above the `dispatch` node, ensuring that it is also evaluated before the arguments of the considered method.

The proof for Law 7 is almost equal to the proof for Law 3. The only difference is that on the evaluation tree for the left hand side, *body′* appears after *body*, and on the right hand side, *body′* appears above the `exit` node. This also ensures that *body′* is evaluated after *body* in both sides of the law.

However, Laws 13, 14, and 15 should be considered differently. The proof for Law 13 would rely on the composition of MCI locations (|| operand on event locations) to ensure that a registered event matches two or more join points. As the only difference between the left hand side and right hand side is the `superimpose` declarations (consequently the registry), both evaluation trees would be equal. According to the MCI semantics, the evaluation of the || operator is the same as evaluating its first operand an then its second operand. Both evaluations register the same piece of code to execute at different events.

The proof for Laws 15 and 14 would rely on removing the `callee` and `caller` constructs respectively. In the MCI semantics, these constructs only bind variables to be used by the advice, they do not constrain the types as occurs with `this` and `target` in AspectJ. As type restrictions are apart from variable binding, we can remove the variable binding given that the variable is not used inside the advice.

We do not discuss the remaining laws formally. As most laws are very simple and intuitive, since each one deals with one construct at a time, their description provides informal arguments describing why the two sides of the laws are equivalent. Hence, we generally described how to map an AspectJ construct to its corresponding Java implementation. Moreover, some laws when applied from right to left, perform a transformation very similar to the transformation applied by the AspectJ compiler to weave aspects and classes.

5. RELATED WORK

This paper uses an existing operational semantics for Method Call Interception [15] to represent aspect-oriented programming laws and reason about them. It seemed appropriate to choose this semantics because of its simplicity, its model of extending an object-oriented language, and its capacity to represent several types of advices from AspectJ.

However, there are other approaches for reasoning about aspect-oriented programs. It would be difficult to represent the laws using most of them. Douence et. al. [6] define a

domain-specific language, along with its semantics, to define crosscuts based no execution monitoring. His system is based on events similarly to the Observer [8] pattern.

Andrews [2] presents process algebras as a formal basis for aspect-oriented languages. He uses a subset of CSP tailored to his purpose, representing join points as synchronization sets. He also defines an equivalence notion between programs and uses it to show the correctness of his weaving process. He uses an imperative language. We use the MCI semantics because it is much simpler and extends the semantics of an object-oriented language just as AspectJ extends Java.

Wand et. al. [19] define a semantic model for dynamic join points. This is not appropriate to our purpose because we needed a semantics in which we could represent AspectJ features. Xu et. al. [20] use a reduction strategy to transform aspect-oriented programs to implicit invocation. This transformation allows them to reason about the programs using already defined semantics for implicit invocation. Aldrich [1] discusses the problem of modular reasoning about aspect-oriented programs. He defines an aspect-oriented language and associated semantics where modular reasoning is possible. Finally, Barzilay et. al. [3] examine call and execution semantics in AspectJ and their interaction with inheritance.

There is also a related work [4] that includes the definition of object-oriented programming laws, an associated semantics, an equivalence notion, and soundness of the laws. Besides, they also prove the relative completeness of their set of laws by defining a normal form and a reduction strategy to transform any program into the normal form.

Hanenberg, Oberschulte and Unland [10] propose some preconditions to apply an object-oriented refactoring in the presence of aspects. Those conditions guarantee a mapping of join points during refactoring, therefore preserving behaviour. They also propose modifications to refactorings such as *Extract Class* [7] in order to make them aspect-aware and therefore respect the preconditions. The second part of Hanenberg, Oberschulte and Unland's research regards refactorings to AspectJ. In fact, they propose some new refactorings from Java to AspectJ. However, they only discuss the refactoring as a whole and the conditions to apply the refactoring. We not only define preconditions but we are able to prove that our transformations preserve behaviour. We also derived the proposed refactorings using the laws, showing that they preserve behaviour.

Analogously, Iwamoto and Zhao [12] proposes modifications to existing refactorings in order to make them aspect-aware. However, it is a superficial discussion. They only show some examples and give some guidelines on how to avoid the aspect effects on the object-oriented refactorings. They also show examples of refactorings from Java to AspectJ. Although, there is no argumentation about necessary conditions to apply the refactorings to ensure that they preserve behaviour. We used the suggested refactorings and derived them as a composition of laws. Hence, we were able to state in which conditions we can apply the refactorings as well.

Finally, there is a related work [14] that discusses aspect-oriented refactorings showing problems when applying object-oriented refactorings in the presence of aspects. It proposes several complex and interesting refactorings and shows clear and easy to understand examples. The laws we are able to prove with the discussed semantics are enough to prove some of his refactorings, for instance the *Extract Method Calls* refactoring.

6. CONCLUSIONS

This paper is a complement to another work on aspect-oriented programming laws [5]. The previous work relied on the simplicity of the laws, which involve only local changes and deal with one AspectJ construct each. Here we show that the laws can be proved sound according to a formal semantics. We show that specifically Law 3 (*Add Before-Execution*). However, other five laws could be chosen.

For that, we use an operational semantics for Method Call Interception [15] where we could represent some of the laws. We also defined an equivalence relation stating the conditions in which two programs behave the same. The proof is based on the evaluation of both programs and then, the analysis of the resulting environments comparing the values of object fields.

However, we can not prove all the laws using this semantics. The MCI semantics is able to represent only `before-call`, `before-execution` and `after-execution returning` advices from AspectJ. Thus, we can only reason about the laws related to those advices. To enable the proof of the remaining laws, we should define a completely new language, along with its semantics, including all the AspectJ constructs covered by the laws. Another solution would be to extend an existing language (i.e MCI) to incorporate the missing constructs. Our current solution allows the proof of Laws 3 - *Add before-execution*, 4 - *Add before-call*, 7 - *Add after-execution returning*, 13 - *Merge advices*, 15 - *Remove target parameter*, and 14 - *Remove this parameter*. The proof of other laws is regarded as a future work.

7. ACKNOWLEDGMENTS

We would like to thank Rohit Gheyi and the other Software Productivity Group members who contributed with comments and suggestions. We also would like to thank the anonymous referees for making several suggestions that significantly improved our paper. This work was supported by CAPES and CNPq, both are Brazilian research agencies.

8. REFERENCES

[1] J. Aldrich. Open Modules: A proposal for Modular Reasoning In Aspect-Oriented Programming. In C. Clifton, R. Lämmel, and G. T. Leavens, editors, *FOAL'04 Proceedings: Foundations of Aspect-Oriented Languages Workshop at AOSD 2004; Technical Report CS Dept., Iowa State Univ.*, Mar. 2004.

[2] J. H. Andrews. Process-algebraic foundations of aspect-oriented programming. In *REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192, pages 187–209. Springer-Verlag, Sept 2001.

[3] O. Barzilay, Y. Feldman, S. Tyszberowicz, and A. Yehudai. Call and Execution Semantics in AspectJ. In C. Clifton, R. Lämmel, and G. T. Leavens, editors, *FOAL'04 Proceedings: Foundations of Aspect-Oriented Languages Workshop at AOSD 2004; Technical Report CS Dept., Iowa State Univ.*, Mar. 2004.

[4] P. H. M. Borba, A. C. A. Sampaio, A. L. C. Cavalcanti, and M. L. Cornelio. Algebraic reasoning for object-oriented programming. *Science of Computer Programming*, January 2004.

[5] L. Cole and P. Borba. Deriving Refactorings for AspectJ. In *Proc. of the 4th International Conference on Aspect-Oriented Software Development (AOSD 2005)*, Chicago, USA, Mar. 2005. ACM Press. To appear.

[6] R. Douence, O. Motelet, and M. Sudholt. A formal definition of crosscuts. In *REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192, pages 170–186. Springer-Verlag, Sept 2001.

[7] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison–Wesley, 1999.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object–Oriented Software*. Addison–Wesley, 1994.

[9] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison–Wesley, second edition, 2000.

[10] S. Hanenberg, C. Oberschulte, and R. Unland. Refactoring of aspect-oriented software. In *4th Annual International Conference on Object-Oriented and Internet-based Technologies,Concepts, and Applications for a Networked World (Net.ObjectDays)*, pages 19–35, Erfurt, Germany, Sept. 2003.

[11] C. Hoare, I. J. Hayes, H. Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey, and B. A. Sufrin. Laws of programming. *Commun. ACM*, 30(8):672–686, 1987.

[12] M. Iwamoto and J. Zhao. Refactoring aspect-oriented programs. In F. Akkawi, O. Aldawud, G. Booch, S. Clarke, J. Gray, B. Harrison, M. Kandé, D. Stein, P. Tarr, and A. Zakaria, editors, *The 4th AOSD Modeling With UML Workshop*, 2003.

[13] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting Started with AspectJ. *Communications of the ACM*, 44(10):59–65, October 2001.

[14] R. Laddad. Aspect-Oriented Refactoring Series. TheServerSide.com, Dec. 2003.

[15] R. Lämmel. A Semantical Approach to Method-Call Interception. In G. Kiczales, editor, *Proc. of the 1st International Conference on Aspect-Oriented Software Development (AOSD 2002)*, pages 41–55, Twente, The Netherlands, Apr. 2002. ACM Press.

[16] M. Monteiro and J. Fernandes. Towards a Catalog of Aspect-Oriented Refactorings. In *4th International Conference on Aspect-Oriented Software Development (AOSD 2005)*, Chicago, USA, Mar. 2005. ACM Press.

[17] W. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, Urbana-Champaign, IL, USA, 1992.

[18] D. Roberts. *Practical Analysis for Refactoring*. PhD thesis, Urbana-Champaign, IL, USA, 1999.

[19] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. In G. T. Leavens and R. Cytron, editors, *FOAL 2002 Proceedings: Foundations of Aspect-Oriented Langauges Workshop at AOSD 2002*, number 02-06 in Technical Report, pages 1–8. Department of Computer Science, Iowa State University, Apr. 2002.

[20] J. Xu, H. Rajan, and K. Sullivan. Aspect Reasoning by Reduction to Implicit Invocation. In C. Clifton, R. Lämmel, and G. T. Leavens, editors, *FOAL'04 Proceedings: Foundations of Aspect-Oriented Languages Workshop at AOSD 2004; Technical Report CS Dept., Iowa State Univ.*, Mar. 2004.