Essential Retroactive Weaving

Robin Salkeld  Ronald Garcia
University of British Columbia
Vancouver, BC, Canada
{rsalkeld,rxg}@cs.ubc.ca

Abstract
To help analyze unexpected behaviour, programming language environments and tools are beginning to support high-fidelity recordings of program executions. Such recordings are typically low-level and difficult to work with directly. Debugging and analyzing these recordings is easier and more powerful if it is possible to simulate executing additional code in the past context of the recording. In prior work we proposed retroactive weaving, the process of evaluating aspects as if they were present during a past execution. This concept is intended as a general framework for introducing additional code and defining the semantics of executing it post-hoc.

In this paper we express retroactive weaving as a transformation on aspect-oriented programming languages and their semantics. We demonstrate this transformation by applying it to a simple core aspect-oriented language, and through a definitional interpreter illustrate its interactions with first-class function values, mutable state, and external input and output. In particular a key concern of retroactive weavers is maintaining soundness: behaving consistently with the context of the past execution, and failing if missing information makes this impossible. Retroactive weavers may need to include extra isolation or runtime checks to meet this requirement.

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

General Terms Languages, Design, Algorithms

Keywords Aspect-oriented programming, execution recording, tracing, replay

1. Introduction
Many modern software systems help developers diagnose errors by recording execution traces that developers can replay and analyze. However, these tools do not provide an intuitive and expressive high-level interface for interacting with traces. We assert that the original programming language is an ideal interface, and that providing the same semantics for its evaluation post-hoc is a powerful and intuitive solution. In our position paper [9], we proposed the concept of retroactive weaving, the process of evaluating aspects as if they were present in a past execution, and described an implementation for an aspect-oriented extension of C. We have also presented evidence for the utility of retroactive execution for various post-hoc analyses via an implementation for the Java Virtual Machine [8]. We have since extended this implementation to create a retroactive weaver for AspectJ.

Our experience with retroactive weaving motivates us to express the concept in abstract, simpler, and more precise terms. Our contribution in this paper is a more essential and general expression of the idea, which should be applicable to any aspect-oriented language with varying effectiveness. Our presentation centres around a simple functional language called RAPL (Retroactive Aspect Programming Language), which serves as a minimally-defined example to illustrate how retroactive weaving interacts with core programming language features. We present the language and its definitional interpreter in three layers: we first present the base functional language, then extend it to support aspects, and then introduce support for retroactive weaving. Finally, we explore a general notion of soundness for retroactive weaving, and explain how our example language and interpreter realize it.

2. Base Language
We first describe the functional core of RAPL, which we intend to reflect the essence of many modern programming language features as simply and unsurprisingly as possible. RAPL has integers, Booleans, and atomic symbols as primitive datatypes. It provides symbols for marking join points and expressing pointcuts (see Section 3). It also includes a void value, which is returned from effectful operations. It supports first-class functions and mutable boxes, using a call-by-value evaluation strategy. Finally, it includes two operations named read and write for external input and output of integers, respectively, as these interact non-trivially with retroactive weaving. Figure 1 contains the RAPL expression grammar; our reference interpreter, which is in the PLAI Scheme dialect of Krishnamurthi [4], contains an equivalent ExprC datatype. The complete interpreter implementation is available at https://github.com/robinsalkeld/rapl.
3. Adding Aspects

We now extend RAPL to include a single aspect-oriented feature: a simple form of pointcuts and advice for function application. AspectJ [3] provides the quintessential example of pointcuts and advice, in which pointcuts quantify sets of join points to affect and advice methods define how the join points are modified. Advice may be declared to run before, after, or around (i.e. in place of) a quantified join point; the latter flavour is the most expressive, and supports a distinguished proceed expression that represents resuming the original join point, or, in the case of overlapping advice, the next advice method in the chain.

In RAPL, the only type of join point supported is the application of functions. Since RAPL has first-class function values, around-style advice can be expressed as higher-order functions that accept and return functions. Within the body of such an advice function, invoking the passed-in function is analogous to the proceed expression in AspectJ, as shown in previous work [2].

Adding aspects to the base version of RAPL requires two new expression cases:

\[ t \in \mathbb{T}_{\text{ERM}}, \quad t := \quad ... \quad | \quad (\text{tag} \ t) \quad | \quad (\text{aroundapps} \ a \ t) \]

We first add a mechanism for tagging values with arbitrary metadata to the language: the expression \((\text{tag} \ t \ e)\) dynamically attaches the value of the \(t\) expression to the value of the \(e\) expression. Tagged values behave identically to untagged values, except that computation involving tagged values can be identified and modified by advice. The tagging construct provides a means of identifying join points, since otherwise function definitions have no external identity. The sub-expressions that are explicitly tagged are used to potentially wrap every tagged abstraction value before it is applied to an argument. Advice takes the form of a function that accepts two arguments: the tag attached to the function being applied, and the untagged form of that function. The result of the advice function is then applied to the argument in place of the original.

Here is an example of an aspect that applies to the factorial function as defined above. This aspect traces the argument and result of each recursive call to the factorial function:

\[
\begin{align*}
(\text{factorial} \ (x)) & := \cdots \\
& | (\text{tag} \ t \ (\text{factorial} \ (x))) \\
& | (\text{aroundapps} \ a \ (\text{factorial} \ (x)))
\end{align*}
\]

Our reference interpreter also uses the following internal definitions, which are referenced in later excerpts:

\[
\begin{align*}
\text{define-type Value} \\
& \quad \text{[numV} \ (\text{n number?})] \\
& \quad \text{[boolV} \ (\text{b boolean?})] \\
& \quad \text{[closV} \ (\text{arg symbol?}) \ (\text{body ExprC?}) \ (\text{env Env?})] \\
& \quad \text{[boxV} \ (\text{l Location?})] \\
& \quad \text{[symbolV} \ (\text{s symbol?})] \\
\text{define-type Binding} \\
& \quad \text{[bind} \ (\text{name symbol?}) \ (\text{value Value?})] \\
\text{define Env?} \ (\text{listof Binding?}) \\
\text{define mt-env empty} \\
\text{define Location? number?} \\
\text{define-type Storage} \ (\text{cell} \ (\text{Location Location?}) \ (\text{val Value?})) \\
\text{define Store?} \ (\text{listof Storage?}) \\
\text{define mt-store empty} \\
\text{define-type Result} \\
& \quad \text{[v\*s} \ [\text{v Value?}) \ (\text{s Store?})] \\
\text{; Top-level evaluation function} \\
& \quad \text{[ExprC} \rightarrow \text{Value} \\
& \quad \text{define} \ (\text{interp-exp expr}) \\
& \quad \text{[v\*s-v} \ (\text{interp expr mt-env mt-store})] \\
\text{; Recursive interpretation function} \\
& \quad \text{[ExprC Env Store} \rightarrow \text{Result} \\
& \quad \text{define} \ (\text{interp expr env store}) \\
& \quad \text{[type-case ExprC expr} \quad ...])
\end{align*}
\]

Since RAPL does not have modules, we encapsulate this aspect as a function that takes a computation which is frozen in a thunk and advises the result of activating it. Common higher-level language features such as top-level definitions and global namespaces would make composing these modules less awkward.
We tag the factorial function so the advice will apply:
(rec (lambda (fact)
  (tag 'fact
    (lambda (x)
      (if (equal? x 0)
        1
        (* x (fact (+ x -1))))))))

We may then combine the modules by applying the aspect function to the factorial prompt thunk:

Listing 4: Possible input and output for applying Listing 3 to Listing 2

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>result:</th>
<th>result:</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; 3</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>&gt; 0</td>
<td>Program result: 0</td>
<td></td>
</tr>
</tbody>
</table>

Both advice and tagging may be nested. The next section provides a concrete implementation of aspect weaving that precisely defines the semantics for both forms of nesting.

4. Aspect Weaving

We now modify the interpreter to implement the semantics of aspects via dynamic weaving; coordinating the cross-cutting concerns as expressions are evaluated. The scope of an aroundapps declaration is the dynamic extent of its expression argument and hence must be tracked dynamically, rather than bound to closures as the environment is. The stack of active advice is therefore maintained in a separate argument to the interpretation function, and is generally passed down through recursive invocations of interp.

Other forms of advice, such as advising operations on boxes, would be defined as additional type cases in the Advice datatype. The implementation of `tag` and `aroundapps`, represented by the ExprC cases `tagC` (tag v) and `aroundappsC` (advice extent) respectively, is trivial:

```scheme
(define-type Advice
  (aroundappsA (advice Value?)))
(define AdvStack? (Listof Advice?))

; ExprC Env AdvStack Store -> Result
(define (interp expr env adv sto)
  (match expr
    [(tag v) (interp-tag tag v env adv sto)]
    [
      (aroundappsC (advice extent)
        (interp-aroundapps advice extent env adv sto))
        ...]
    [else (error "only functions can be applied")]))

; ExprC ExprC Env AdvStack Store -> Result
(define (weave adv f sto)
  (match adv
    [
      (aroundappsA (g)) (apply g (list tag f) adv sto)]
    [
      (aroundappsC (advice extent)
        (interp-aroundapps advice extent env adv sto))
        ...])
    [else (error "only functions can be applied")])

; ExprC ExprC Env AdvStack Store -> Result
(define (weave-for-tag adv tag f sto)
  (match adv
    [
      (aroundappsA (g)) (apply g (list tag f) adv sto)]
    [
      (aroundappsC (advice extent)
        (interp-aroundapps advice extent env adv sto))
        ...])
    [else (error "only functions can be applied")])

; Wrap aroundapps to apply advice
(define (apply-with-weaving f args adv sto)
  (match adv
    [(aroundappsA (g)) (apply g (list tag f) adv sto)]
    [(aroundappsC (advice extent)
        (interp-aroundapps advice extent env adv sto))
        ...])
    [else (error "only functions can be applied")])

The only other case directly affected is applications. The base implementation first reduces the function and the arguments to values, then invokes the apply routine to evaluate the function body with the augmented environment:

```scheme
(define (apply f args adv sto)
  (match f
    [(type-case Value f
cron (params body env))
      (let ((bs (map bind params args)))
        (interp body (append bs env) adv sto))
      [else (error "only functions can be applied")])
  (match args
    [(type-case Result (interp v env adv sto)
cron (v-w s-w)
      (v*s (v-w s-w)))
      [else (v*s f sto)])
    [else (error "only functions can be applied")]))

; Applies all advice in scope for all tags on f
(define (weave-advice adv tag advice f sto)
  (match adv
    [(aroundappsA (g)) (apply g (list tag f) adv sto)]
    [(aroundappsC (advice extent)
        (interp-aroundapps advice extent env adv sto))
        ...])
    [else (error "only functions can be applied"))])

; Applies all advice in scope for a single tag on f
(define (weave-for-tag adv tag f sto)
  (match adv
    [(aroundappsA (g)) (apply g (list tag f) adv sto)]
    [(aroundappsC (advice extent)
        (interp-aroundapps advice extent env adv sto))
        ...])
    [else (error "only functions can be applied")])

; Apply a single advice function to f
(define (weave-advice adv tag advice f sto)
  (match adv
    [(aroundappsA (g)) (apply g (list tag f) adv sto)]
    [(aroundappsC (advice extent)
        (interp-aroundapps advice extent env adv sto))
        ...])
    [else (error "only functions can be applied")])

Nestad advice declarations and tags are handled with a double iteration: first over nested tags from innermost to outermost, and for each tag over nested advice declarations from inner to outer. Any tags attached to values do not affect their semantics outside of advice; for every other operation in the language the tags on each operand are stripped before performing the original logic.

5. Retroactive Aspects

To define the semantics of retroactive aspects precisely we relate retroactive weaving, which is the process of executing them, to execution in the presence of conventional weaving for AOP languages in general. We use mathematical notation to express the key requirement.

Let E be the partial function that represents evaluating a program to produce observable results, which is undefined for evaluations that do not terminate. For impure languages with external input/output, E depends not only on a term but its context, which we denote with ∈ CTXT. The range of E is OBS; for RAPL this is the result value and any output. Therefore E : POM ◯ CTXT → OBS.

In the spirit of Masuhara and Kiczales [5], we model AOP in general via a weaving process with signature W : ASPECT ×
PGM → PGM. Therefore executing an aspect a together with a program p is modelled as \(E(W(a,p),c)\).

We augment E to also produce a trace, an abstract notion of intermediate computation information:
\[E_{\text{traced}} : \text{PGM} \times \text{CTXT} \rightarrow \text{OBS} \times \text{TRACE}.\]

Retroactive weaving is another evaluation function \(RW : \text{ASPECT} \times \text{TRACE} \rightarrow \text{OBS}\). Its behaviour is defined by an invariant: for any program \(p\), aspect \(a\) and execution context \(c\), if \(E_{\text{traced}}(p,c) = (o,t)\), and \(RW(a,t)\) is defined, then \(RW(a,t) = E(W(a,p),c)\).

The intuition behind this model is that a trace encodes partial information about the past execution context. Retroactive weaving produces the observable behaviour the augmented program would have produced in that context. If the augmented program depends on missing information, the process must signal an inconsistency error, represented here by undefinedness in RW.

### 6. Retroactive Weaving

We now describe how our definitional interpreter provides both tracing and retroactive weaving, the process of executing retroactive aspects.

#### 6.1 Recording and Reading Traces

The augmented interpreter must be able to record the relevant information during one execution and read this information during a future execution. Thus we first require interpretation to record a subset of the states it reaches while interpreting externally, so that retroactive weaving can identify join points post-hoc and apply advice as required. Within this context we use trace to refer to an ordered list of recorded interpretation states.

To consume traces one state at a time during retroactive execution, we add another parameter to the interpretation function for the remaining trace to read. The head of this trace represents the current state of the original execution; under normal evaluation this list will be empty. The retroactive weaving process reacts to each recorded state, potentially performing additional execution before moving to the next recorded state by popping the head of the trace list.

```scheme
; ExprC Env AdvStack Store Trace -> Result
(define (interp expr env adv sto tin)
  ...

(define Trace? (listof State?))
(define mt-trace empty)

; Trace -> State
(define (trace-state tin)
  (first tin))

; Trace -> Trace
(define (next-trace-state tin)
  (rest tin))

To produce traces, we extend the Result data type so that every computation can also provide the trace for that computation. In addition, during retroactive weaving the input trace must be threaded through the interpreter much as the store is, and hence we also extend Result to include the remaining input trace:

```scheme
(define-type Result
  (v*s*t*t (v-w s-w tin-w tout-w)
    (type-case Result (apply v-w args adv s-w tin-w)
      (type-case Result (apply v-w args adv s-w tin-w)
        ...

The datatypes that define our version of tracing are as follows:

```scheme
(define-type Control
  [app-call (f Value?) (args (listof Value?)
    (tin Trace?) (tout Trace?)
  )]

(define-type Result
  (v*s*t*t (v-w s-w tin-w tout-w)
    (type-case Result (apply v-w args adv s-w tin-w)
      (type-case Result (apply v-w args adv s-w tin-w)
        ...

The Control datatype enumerates the different kinds of interpretation control flow we record, in particular just before applying a function value to an argument value, and just after such a call produces a result value. The State datatype combines the interpreter control point with the interpreter arguments. It does not include the environment because advice in RAPL is modelled with function calls, and hence cannot access the lexical scope of join points, but this would not necessarily be true in AOP languages that make use of dynamic binding in aspects.

Since our traces only carry information about function applications, only the main application routine extends the current trace:

```scheme
(define (apply-with- weaving f args adv sto tin)
  ...

(define-type AdvEnv Store Trace? -> Result
  (type-case Result (trace-state tin)
    ...

Other operations in RAPL could be recorded in the same style; in general these traces are produced by appending the sub-traces for individual sub-computations together in evaluation order.

We omit the details of serializing these traces to and from persistent storage, in our case one file per trace, as they do not affect the semantics of retroactive weaving. In practice, however, there is ample opportunity for optimizing the writing and reading of such traces, and scalable implementations can be quite sophisticated [6]. In particular, recording a full copy of the store at every event can be expensive, and more practical implementations will instead record individual changes to the store incrementally.

In all but the simplest of programming languages and their environments, it will not be possible or feasible to record all of the information a program could have queried during its execution. This is especially true of more mainstream languages that have access to file systems, networks, and more unpredictable sources of values such as the current time. The particular instance of tracing we present here is chosen to be simple and sufficient to support a reasonable number of retroactive aspects, and we intentionally omit many other interpreter states as well as the external input accessed via read during execution. For a more complete discussion of how this affects our implementation’s completeness see Section 7.

#### 6.2 Retroactive State

The state of the retroactive interpretation can build on the original interpretation state. In particular retroactive execution can use references to values, including store locations and their contents, from the original execution. Consider an alternative version of the factorial function which uses a box internally to track its counter:

```scheme
(let ((fact_helper
  (lambda (bx)
    (let ((x (unbox bx)))
      (if (equal? x 0)
        1
        (let ((s (set-box! bx (+ x -1)))
          (+ x (fact bx))))))))))

Listing 5: An alternate factorial implementation using boxes

To create an equivalent version of Listing 3 for this implementation, the advice needs to dereference the box passed to the helper function in order to obtain the actual value of x. Therefore the
retroactive weaving interpretation must deal with a mix of locations: new locations created during the retroactive evaluation and old locations from the trace. In addition, the values stored in old locations may change as the trace is traversed, so references to old locations must somehow be kept current. Finally, it is necessary to distinguish old and new locations to detect inconsistent executions (see Section 7).

Our approach is to add another case to the Value datatype to implement a layer of indirection on values obtained from the trace. This aligns closely with how production implementations are likely to be implemented, as it allows the underlying trace and its store to proceed independently of the retroactive state [8].

When a value from prior state is bound by a retroactive aspect (such as the box passed to advice for the fact_helper function above), it must be lifted to the retroactive context, so that new and old store locations can be distinguished. The value may be a box itself, or it may be a compound value such as a closure which may transitively refer to store locations. We define a lift-trace-value function in our interpreter for this purpose. The omitted Value cases are handled by straightforward structural recursion: primitive values are untouched, and tagged values and the value bound by the environments stored in closures are lifted piecewise.

; Value -> Value
(define (lift-trace-value v)
  (type-case Value v
    (boxV (trace-loc)
      (traceValueV v))
    ...))

We then augment fetching from the store to handle boxes from the trace:

; Store Trace Value -> Value
(define (fetch sto tin b)
  (type-case Value b
    [boxV (loc) ...] ; As before
    [traceValueV (v)
      (type-case State (trace-state tin)
        [state (c adv s-t tin-t)
          (traceValueV v)
          [else (error "interp "attempt to unbox a non-box")]]))

6.3 Retroactive Control
Implementing retroactive weaving involves producing the extra execution that an aspect specifies at various positions in the trace. When an application callback is applied retroactively to an application in the trace, we need to use a placeholder to resume the original execution - that is, reading the rest of the trace - instead of evaluating the application. To achieve this we add another case for values:

(define-type Value ...
  [resumeV])

Any tags on the original function must be carried over to the stub value, so that application advice will behave identically:

; Value -> (listof Value)
(define (all-tags v)
  (type-case Value v
    [taggedV (tag tagged)
      (cons tag (all-tags tagged))
      [else empty]))
  [listof Value] Value -> Value
(define (deep-tag tags v)
  (foldr taggedV v tags))

Applying this value as if it were a function instead resumes the process of weaving the trace:

; Value (listof Value) AdvEnv Store Trace -> Result
(define (apply-without-weaving f args adv sto tin)
  (type-case Value f
    [valueV (params body env)
      (let (([bs ([map bind params args]]))
        (interp body (append bs env) adv sto tin))]
      [resumeV () (rw-call f args adv sto tin)]
      [else (error "only abstractions can be applied")])

The core of the retroactive weaving implementation are these three mutually recursive functions:

; Value (listof Value) AdvEnv Store Trace -> Result
(define (rw-call f args adv sto tin)
  (type-case State (state-c (trace-state tin))
    [app-call (f args)
      (type-case Result (rw-replay-call f args adv sto tin)
        (v*s*t	 v-r s-r tin-r tout-r
         (rw-result adv s-r
           (next-trace-state tin-r))))]
      [app-result (r)
        (v*s*t	 (lift-trace-value r)
          sto tin mt-trace)])
  rw-replay-call consumes the next sub-sequence of the trace from a function application up to its corresponding result, and rw-result continues to consume such sub-sequences until it reaches the result for the current application. These routine essentially reconstruct the original tree of recursive calls to the interpretation function. Note that these versions of the core retroactive weaving routines do not produce a trace for retroactive weaving itself in order to simplify presentation, but adding this tracing using the implementation strategy shown in Section 6.1 is straightforward.

The top-level entry point to retroactive weaving is a separate but related interp-rw function, which corresponds to RW in the abstract model in Section 5. As demonstrated above, in RAPL advice can be declared in discrete modules if the computation they advise is provided as a parameter, delayed within a thunk. The resumeV value is also used to represent the trace as such a thunk.

7 Ensuring Soundness
The implementation so far will behave correctly for many retroactive aspects. However, not all retroactive aspects are sound according to the semantics defined in Section 5; if the augmented program attempts to access information that was not recorded, retroactive weaving is required to terminate in an error, whereas the interpreter thus far may instead produce inconsistent observable behaviour.

The implementation above assumes that when a recorded state has been processed and the paused original execution resumed, that original execution would have reached the same next recorded state in the trace. If a retroactive aspect perturbs the program state in some way, the program may have continued to make an unsupported operation as above, so we cannot assume this is safe. There-
fore, for this particular implementation of tracing and weaving, ensuring soundness is equivalent to ensuring that retroactive advice would not have perturbed the original execution.

Since aspects in RAPL are quite general and expressive, there are several ways that retroactive weaving can fail:

New external side-effects: Advice itself might attempt to add additional interaction with the original context. In RAPL this means extra calls to `read`, which conceptually consume values from the program input prematurely and shift the values read by the original program, leaving the later inputs uncertain. This is preventable by replacing the source drawn by the `read` expression with a stub that raises an error during retroactive weaving.

Modifying arguments: Advice may pass a different list of arguments to the wrapped function than was originally provided. To prevent this, we add a comparison of the arguments the stub `resumeV` value is applied to against the arguments provided in the original join point, within the `rw-call` function.

Modifying results: Similarly, advice may return a different result than a join point of the original computation. Another check must be inserted before returning from `rw-replay-call` to compare the value produced by the advice back stack to the original.

Modifying the original store: Advice could also perturb the original execution more indirectly by mutating boxes, so we modify the implementation of `set-box!` to raise an error if the given box is a reference to the original store (i.e. a `traceValueV` as described in Section 6.2).

Modifying control flow: More deviously, advice may fail to invoke advised functions, or invoke them more than once. Because the construct that represents proceeding in advice is a first-class value (i.e. a function), it could also be bound and applied later, outside the scope of the advice. All these cases can be prevented by attaching the length of the remaining trace to the stub `resumeV` value at the time it is created, and comparing this to the current length of the remaining trace in the store value whenever it is applied. This ensures each stub is applied in order and no more than once. An additional check after the top-level call to `rw-replay-call` to verify that the entire trace has been consumed ensures that each stub is applied at least once.

7.1 Deterministic Replay

The restrictions above depend heavily on the exact information recorded during the original execution. Rather than recording the full state of interpretation at relevant points, which can be very expensive, the runtime could instead record non-deterministic events, so that the state can be reconstructed by replaying the interpretation. For RAPL, recording would become storing only the sequence of input integers, and replaying would involve assigning the read source to be that sequence.

The straightforward approach to retroactive weaving using replay is to trace the replay process and then use retroactive weaving on the trace as above. This could be made more efficient by having the replay process produce a stream of states which the weaver consumes. It is tempting to optimize this further by directly weaving the retroactive aspects against the original program during the replay interpretation instead. This is not sound in general, though, without again modifying the interpreter to guard against new retroactive external side-effects as above.

8. Related Work

RAPL bears a strong resemblance to AspectScheme [2], another aspect-oriented language with first-class function values. The key place they differ is that AspectScheme is an AOP extension to an existing full-featured programming language, whereas RAPL is intended to be a core language, with the minimum features required to support retroactive weaving. Some of AspectScheme’s features, such as statically scoped advice and equality of functions via source location, can be expressed via desugaring to RAPL.

De Fraine et. al. provide a core calculus for AOP with their A calculus [1]. The A calculus is object-oriented, but like RAPL also models proceed as binding a closure-like value, and supports passing closures as first-class values. Because the RAPL interpreter is more focused on modelling two alternative strategies of aspect weaving, it avoids object-orientation and types to keep the semantics simpler.

9. Conclusion

We have presented the concept of retroactive weaving as an abstract concept directly related to the semantics of conventional aspect weaving for aspect-oriented programming languages. We provided a definitional interpreter that implements retroactive weaving for a simple core language that illustrates the interaction of retroactive weaving with common core language features. Finally, we discussed the soundness requirement for such an implementation and its consequences.

There are several dimensions of future work that can build on this first step. It should be possible to apply the general model defined in Section 5 to AOP languages beyond those based on pointcuts and advice, such as crosscutting contracts as in AspectJML [7]. It would also be instructive to investigate static analyses to check for valid retroactive aspects and avoid the runtime errors outlined above.

Acknowledgments

A portion of this work was funded by an NSERC Discovery Grant.

References


