**Retroactive Aspect Weaving in Miraj**

**Base language**

The base language is extremely similar to the example language with mutable variables presented in [S. Krishnamurthi 2007, Section 8.2] (<http://cs.brown.edu/courses/cs173/2012/book/mut-struct-vs-var.html#(part._mut-vars>). However, the language is procedural rather than functional, in the style of Chapter 6 (<http://cs.brown.edu/courses/cs173/2012/book/From_Substitution_to_Environments.html>). This is because the quintessential AOP pointcut mechanism identifies joinpoints using function names, which are more dubious in a functional language.

The expression datatype is:

(define-type ExprC

[numC (n number?)]

[varC (s symbol?)]

[appC (fun symbol?) (arg ExprC?)]

[plusC (l ExprC?) (r ExprC?)]

[multC (l ExprC?) (r ExprC?)]

[setC (s symbol?) (v ExprC?)]

[letC (s symbol?) (val ExprC?) (in ExprC?)]

[defC (d defC?) (in ExprC?)]

[seqC (b1 ExprC?) (b2 ExprC?)]

[ifZeroOrLessC (c ExprC?) (t ExprC?) (f ExprC?)]

[writeC (l string?) (v ExprC?)]

[readC (l string?)]

)

Note the writeC and readC expressions, for reading program input and writing program output, are included to ensure non-deterministic behaviour is modeled somehow. The read behaviour is pluggable, in that the interpreter delegates the readC expression to a boxed lambda:

(define read-source (box (lambda () (string->number (read-line)))))

The need for this will be covered later on.

The interpretation function’s signature and some related datatypes:

(define-type DefC

[bindC (name symbol?) (loc Location?)]

[funC (name symbol?) (arg symbol?) (body ExprC?)])

(define Env? (curry andmap DefC?))

(define-type Storage

[cell (location Location?) (val Value?)])

(define Store? (curry andmap Storage?))

(define-type Result

[v\*s (v Value?) (s Store?)])

; The usual recursive interpretation function

(define (interp [expr ExprC?] [env Env?] [sto Store?]) Result?

...)

(define (interp-program [exp ExprC?])

(interp exp empty empty))

Note that a Miraj program is simply an expression to evaluate. The only types of values are integers, and the fact that procedures must accept and return an integer makes for some slightly odd examples, but simplifies the implementation.

We will use an interactive factorial calculator as a running example. Here is the source in Miraj syntax (presume this is stored in the file fact.rkt).

int fact(int x) {

if (x <= 0) {

return 0;

} else {

return x \* fact(x – 1);

}

}

int fact\_loop(int x) {

int in = read();

if (in <= 0) {

return in;

} else {

write(fact(in));

fact\_loop(x);

}

}

fact\_loop(42);

When run, this program repeatedly accepts a number on standard in and prints out the factorial of that number, terminating when 0 or a negative number is entered. Note the value passed to fact\_loop is arbitrary. The same example using the above AST is below.

(miraj

(list (fdC 'fact 'x

(ifZeroOrLessC (varC 'x)

(numC 1)

(multC (varC 'x)

(appC 'fact (plusC (varC 'x) (numC -1))))))

(fdC 'fact\_loop 'x

(letC 'in (readC)

(ifZeroOrLessC (varC 'in)

(varC 'in)

(seqC (writeC (appC 'fact (varC 'in)))

(appC 'fact\_loop (varC 'in)))))))

empty

(appC 'fact-loop (numC 42))

)

**Aspect language**

We then add a single aspect-oriented feature: a very simple variation of pointcuts and advice.

(define-type AdviceDefC

[aroundC (name symbol?) (arg symbol?) (body ExprC?)])

(define AdvEnv? (curry andmap AdviceDefC?))

Only around advice is supported since before and after advice definitions are expressible using around advice. In this case the term joinpoint is synonymous with procedure call, and the only pointcut supported is matching procedure names exactly.

We add one more case to the expression syntax: proceedC.

(define-type ExprC

[numC (n number?)]

[varC (s symbol?)]

[appC (fun symbol?) (arg ExprC?)]

[plusC (l ExprC?) (r ExprC?)]

[multC (l ExprC?) (r ExprC?)]

[setC (s symbol?) (v ExprC?)]

[letC (s symbol?) (val ExprC?) (in ExprC?)]

[seqC (b1 ExprC?) (b2 ExprC?)]

[ifZeroOrLessC (c ExprC?) (t ExprC?) (f ExprC?)]

[writeC (v ExprC?)]

[readC]

[proceedC (v ExprC?)]

)

The proceed expression is only valid within advice bodies, and applying it to an expression passes that value into the original function (or the next advice in the chain, when multiple advice declarations match for the same joinpoint). Invoking proceed in a function body leads to a runtime error.

Here is an example of a tracing advice definition for the factorial program above (presume this is stored in the file fact\_advice.rkt).:

int around(int y) : call(fact) {

int result = proceed(y);

write(y);

write(result);

return result;

}

And again in the AST datatype:

(aroundC 'fact 'y (letC 'result (proceedC (varC 'y))

(seqC (writeC (varC 'y))

(seqC (writeC (varC 'result))

(varC 'result)))))

**Inline Weaving**

The signature of the interp function now has to include the advice environment and the current continuation bound to proceed:

(define (interp [expr ExprC?] [env Env?]

[fds FunEnv?] **[ads AdvEnv?]**

[sto Store?]

**[proceed procedure?]**) Result?

...)

Most of the interpretation cases are the same, modified only to pass along the advice environment and the proceed binding. Only procedure application is interesting, which we cover in detail below.

First, a convenience method for interpreting an expression with a single new symbol bound to a value. This will be used by both procedure and advice evaluation (as well as letC).

(define (interp-with-binding [expr ExprC?] [name symbol?] [a Value?]

[env Env?] [fds FunEnv?] [ads AdvEnv?]

[sto Store?] [proceed procedure?]) Result?

(let ([where (new-loc)])

(interp expr

(extend-env (bind name where) env)

fds

ads

(override-store (cell where a) sto)

proceed)

))

The core implementation of advice semantics is the apply-advice function, which maps continuations to continuations, adding the advice behaviour if the pointcut matches (i.e. if the function name matches).

(define (apply-advice [n symbol?] [fds FunEnv?] [ads AdvEnv?]

[advice AdviceDefC?] [proceed procedure?]) procedure?

(type-case AdviceDefC advice

[aroundC (name param body)

(cond

[(symbol=? n name)

(lambda (val sto)

(interp-with-binding body param val empty fds ads

sto proceed))]

[else proceed])]))

This particular approach will prove to simplify retroactive weaving as well as inline weaving.

The call-closure function provides the base continuation for invoking the procedure:

(define (no-proceed (val Value?) (sto Store?)) Result?

(error 'no-proceed "proceed called outside of advice"))

(define (call-closure [fd FunDefC?] [fds FunEnv?] [ads AdvEnv?]) procedure?

(lambda (val sto)

(interp-with-binding (fdC-body fd) (fdC-arg fd) val empty fds ads

sto no-proceed)))

Weaving at a particular joinpoint becomes folding the apply-advice function over the advice environment:

(define (weave [n symbol?] [fds FunEnv?] [ads AdvEnv?] [proceed procedure?])

(foldr (lambda (val sto) (apply-advice n fds ads val sto)) proceed ads))

The interpretation case for applications is therefore:

[appC (f a) (type-case Result (interp a env fds ads sto proceed)

[v\*s (v-a s-a)

(let\* ([fd (get-fundef f fds)]

[cc (call-closure fd fds ads)]

[woven-closure (weave f fds ads cc)])

(woven-closure v-a s-a))])]

For convenience when weaving inline, and to help define the semantics of retroactive weaving, we need a concept of concatenating programs together:

(define-type MirajProgram

[miraj (fds FunEnv?) (ads AdvEnv?) (exp ExprC?)]

)

(define mt-program (miraj empty empty (numC 0)))

(define (append-programs [p1 MirajProgram?] [p2 MirajProgram?])

;; TODO-RS: Should verify that the environments are mutually exclusive

(miraj (append (miraj-fds p1) (miraj-fds p2))

(append (miraj-ads p1) (miraj-ads p2))

(seqC (miraj-exp p1) (miraj-exp p2))))

Note that advice programs will generally not define a meaningful expression to evaluate; they will normally be appended together with a program that does. For example:

racket miraj\_interpreter\_cli.rkt fact\_advice.rkt fact.rkt

>> 3

0

1

1

1

2

2

3

6

6

>> 5

0

1

1

1

2

2

3

6

4

24

5

120

120

>> 0

**Retroactive Weaving**

Retroactive weaving is the process of evaluating aspects with respect to a previous program evaluation. Here is how the process should look:

racket miraj\_interpreter\_cli.rkt --trace trace.txt fact.rkt

>> 4

24

>> 0

racket miraj\_interpreter\_cli.rkt --weave trace.txt fact\_advice.rkt

0

1

1

1

2

2

3

6

4

24

Thus we require interpretation to record the joinpoints encountered externally, so that a new evaluation endpoint can read those joinpoints and apply advice as needed.

First we modify the procedure handler to record joinpoints:

(define-type JoinPoint

[call (name symbol?) (a Value?) (sto Store?)]

[return (name symbol?) (result Result?)])

(define interp-jps (box '()))

(define (record-interp-jp (jp JoinPoint?))

(set-box! interp-jps (cons jp (unbox interp-jps))))

(define get-interp-jps

(lambda () (reverse (unbox interp-jps))))

(define (call-closure [fd FunDefC?] [fds FunEnv?] [ads AdvEnv?]) procedure?

(lambda (val sto)

(let\* ([\_ (record-interp-jps (call (fdC-name fd) val sto))]

[result (interp-with-binding (fdC-body fd) (fdC-arg fd)

val mt-env fds ads sto no-proceed)]

[\_ (record-interp-jps (return (fdC-name fd) result))])

result

)))

Assume that the interp-jps list is written out to the requested file when interpretation finishes. The retroactive weaving process will then read in this trace and execute as follows:

(define (retroactive-weave [jps list?]

[original-program MirajProgram?]

[new-program MirajProgram?])

(type-case MirajProgram (append-programs original-program new-program)

[miraj (fds ads exp)

(retroactive-weave-box (box jps) fds ads empty))

(define (retroactive-weave-box [jps box?] [fds FunEnv?] [ads AdvEnv?]

[advice-store Store?])

(cond

[(empty? (unbox jps)) empty]

[else

(type-case JoinPoint (list-box-pop! jps)

[call (name arg jp-store)

(let\* ([proceed (lambda (val sto)

(retroactive-weave-box jps fds ads sto))]

[result ((weave name fds ads proceed) arg advice-store)])

(retroactive-weave-box jps fds ads (v\*s-s result)))]

[return (name result) (v\*s (v\*s-v result) advice-store)])]))

This algorithm is essentially reconstructing the call stack of the original program, and using the trace reading process as the continuation for advice to apply to.

Note that the store threaded through the advice execution is completely independent of the stores recorded in the execution trace. This is because there is no language mechanism for the advice to access the original store; the only binding mechanism, capturing procedure arguments, has pass-by-value semantics.

**Retroactive Side-effects**

This seems to be a complete implementation at this point. However, not all executions will be sound according to the semantics of retroactive weaving.

Consider this ill-advised advice – note the change in bold:

int around(int y) : call(fact) {

int result = proceed(y);

write(**read()**);

write(result);

return result;

}

Clearly this will likely produce different results between inline and retroactive weaving, since it alters how the program consumes external state, and will end up prompting for extra input in the retroactive case. In this particular example the error is obvious, but in general the side-effects could be subtle and lead to incorrect conclusions about the original execution.

Therefore retroactive weaving is only valid for a subset of aspects, and the weaving process must produce runtime errors if it cannot reproduce the same semantics as inline weaving. For this particular language this means raising an error if any code executed in the control flow of retroactive weaving attempts to read input. In our implementation this means we have to stub out the read source as follows.

(define (retroactive-weave [jps list?] [fds FunEnv?] [ads AdvEnv?])

(begin (set-box! read-source (lambda ()

(error 'retroactive-side-effect

"cannot call read in retroactive advice")))

(retroactive-weave-box (box jps) fds ads mt-store)))

**Deterministic Replay**

Rather than recording each joinpoint and the state of the interpretation at that point explicitly, we can instead only record non-deterministic events, so that joinpoints can be found by replaying the interpretation. For Miraj, this means recording an interpretation just becomes recording the sequence of input integers, and replaying just involves assigning the read source to be that sequence.

The straightforward approach to retroactive weaving using replay is to record the sequence of joinpoints during replay and then use the same algorithm as above. This could be made more efficient by having the replay process produce a stream of joinpoints which the weaver consumes.

It is tempting to optimize this further by just applying inline weaving of the new advice during the replay interpretation instead. This is not sound in general, though, as illegal advice such as the above example that reads extra input may cause the replay to diverge from the original path.

**Notes/Open Questions**

1. How should I modify the base language in order to illustrate the interaction of old and new state less trivially? One option is to introduce global variables, since that would make stateful advice possible (e.g. tracking the call depth), but adding yet another parameter to most functions is not very appealing.  
     
   Another option is to revert to boxes rather than variables. This lines up less well with Java syntax since primitives are not passed by reference, but does line up with Java semantics since objects are.
2. Under the topic of side-effects, I should also be checking that retroactive advice does not change argument values, change return values, or call proceed less or more than exactly once. I still have to implement this.
3. The example of illegal advice side-effects is not terribly convincing. I am tempted to add a randomC expression and use that instead, as it’s more convincing that someone using this might not realize a call to something like random() is buried somewhere in the advice’s control flow.