AspectScheme – Aspects in Higher-Order Languages

Christopher J. Dutchyn

Computer Science
University of Saskatchewan
dutchyn@cs.usask.ca

AspectScheme is an implementation of the Scheme programming language [4] built on MzScheme [3], providing support for pointcuts and advice aspect-oriented programming. In order to use it, place

```
#lang racket
(require (planet dutchyn/aspectscheme:1:0/aspectscheme))
```

in your code, before using any AspectScheme features[1].

AspectScheme is a lightweight language, constructed using the continuation marks [2] and macro facilities offered by MzScheme. It considers procedure application (`call?`) and procedure-body execution (`exec?`) as *join point*s, within a context of in-progress join points[2]. An additional join point kind, `adv?`, is used to identify join points which are calls to advice bodies. This has utility to avoid introducing unexpected infinite recursion in advice bodies, but even more useful for applying advice to advice.

These join points in context are recognized by *pointcut*s: predicates over the join point and context (represented as lists of posterior and anterior join points). When a pointcut matches, it returns a list of selected values, which are used as arguments to the advice. Primitive recognition pointcuts, (`call?`, `exec?`, and `adv?`), do not return any bindings and can be used simple for join point selection. Other primitive pointcuts, (`target`, `args`) return the procedure and the arguments (respectively) at the join point, so they can be made available to the advice.

```
call?      exec?    adv?    ;kind recognition
target     args             ;binding
top        bottom           ;context bounds
```

Because pointcuts are user-level procedures, they can be combined to provide precise join point selection. There are several standard patterns, and these are provided as utility pointcut combinators:

```
call       exec     adv     ;kinds with binding
&&         ||               ;logic combination
!                           ;logic negation
```

[1] This has been verified with versions up to Racket 5.3.

[2] Arguably, these are not the only principled and nominal occurrences in a Scheme program, but the other obvious candidate `set!` is frowned upon.

The variadic

$$\texttt{(\&\& <pc}_1\texttt{> <pc}_2\texttt{> ...)}$$

combinator concatenates values extracted from the matched `<pc`$_n$`>`. The variadic

$$\texttt{(|| <pc}_1\texttt{> <pc}_2\texttt{> ...)}$$

combinator short-circuits to the bindings from the first match; programmers are recommended to bind the same number of values in each `pc`$_n$, or exercise extreme caution. The unary negation combinator (`(! <pc>)`) throws away any bindings. In order to maintain these consistent-length binding lists, two utility binding pointcuts,

- `(with-args <val> ...)` injects values into binding lists
- `(some-args <bool list>)` selects only a subset of the bindings using a boolean mask.

In the simplest example of binding and combinators, `call` is simply `(&& call? args)`.

Pointcuts receive the join point in context: initially with the anterior join-point list empty, and the posterior join-point list showing the entire stack of join points back to program start. With simple shifting pointcut transformers, we can shift the focus of a pointcut up (`above`) or down (`below`) the join-point stack. These focus-shifting transformers are combined into the expected search the context upward (`cflowbelow`) and downward (`cflowabove`). In addition, a search may be bounded above or below by `cflowtop` and `cflowbottom` respectively. All of these contextual pointcut combinators rely on two primitive pointcuts, primitive `top` and `bottom` to recognize absolute bounds.

```
below      above          ;focus shift
cflowbelow cflowabove     ;search up/down
cflowtop   cflowbottom    ;bound search up/down
```

Finally, there are two utility pointcut transformers which capture the usual facilities in other aspect languages. `(cflow <pc>)` is simply `(&& <pc> (cflowbelow <pc>))`. `(within <pc>)` ensures lexical containment by ensuring no intervening join points (i.e. procedure calls).

```
cflow      within         ;historical convenience
```

Advice transforms the join point into a new procedure which can invoke the original join point execution as `proceed`[3]. Advice is a higher-order procedure, matching the template

```
(lambda (proceed)
   (lambda bindings
      <body>))
```

which may allow the join point (procedure) to proceed zero, one, or more times, by calling it as `(proceed <a`$_1$`> <a`$_2$`> ...)` with new values for the arguments.

[3] Earlier versions called this `jp`, but `proceed` is more common in AOP.

Aspects, consisting of a pointcut and an advice are introduced by the `around` and `fluid-around` constructs:

```
    (around <pc> <adv>      (fluid-around <pc> <adv>
      <body>...)              <body>...)
```

which differ in the scoping of the aspect application. Static aspects (`around`) apply only to join points present lexically within the body. To wit, the join point matching `<pc>` must have the actual call present in the `<body>`, not called deeper in the call hierarchy. Dynamic aspects (`fluid-around`) apply to join points that are present dynamically within the execution of the body. Last, top-level aspects can be installed at top-level (e.g. in the REPL) with

```
    (top-level-around <pc> <adv>)
```

There is no body to be advised, because it is the remainder of the top-level scope. There is no way to remove top-level aspects, save restarting the REPL.

For convenience, a number of standard aspect shapes are supplied; the keyword below replaces `around`. These come in static (unadorned), dynamic (prefixed by `fluid-`), and `top-level-` variations.

1. `before`: execute `<adv>`, then `proceed`
2. `after`: `proceed`, then execute `<adv>`, returning the value from the `proceed`
3. `after-throwing`: `proceed`, and execute `<adv?` only if an exception is thrown[4]

Users should be aware that aspects can break tail-call properties of the join point; a trivial example is tracing function calls. As expected, this will cause stack growth. Pointcuts can also break tail-call properties by forcing continuations to be marked in order to preserve calling context. We adopt a simple approach to continuation marks, and hence advised code may not remain tail-call optimized.

For usage examples, please see `tests.ss` and `tests2.ss` available at the Planet repository. For more details, please refer to [6] or [5]. But, beware, this is an extended implementation, where pointcuts are expected to return arguments from the matches.

We also include an extended version of let/let*/letrec that accepts the MIT-style curried lambdas, just like define. This makes advice much easier to write: it looks like

```
(let ([((<adv> proceed) bindings ...) body...])
  (around ...))
```

This was shamelessly appropriated from Eli Barzilay's `swindle/base.ss` [1].

References

[1] E. Barzilay. Swindle. Internet. http://barzilay.org/Swindle.

[2] J. Clements, M. Flatt, and M. Felleisen. Modelling an algebraic stepper. In *LNCS*, number 2028. 2001.

[3] M. Flatt. *PLT MzScheme: Language Manual*. Rice University, 1997.

[4] R. Kelsey, W. Clinger, and J. Rees. Revised^5 report on the algorithmic language scheme. *HOSC*, 11(1), 1998.

[5] D. Tucker and S. Krishnamurthi. Pointcuts and advice in higher-order languages. In *AOSD*, 2003.

[6] D. Tucker, S. Krishnamurthi, and C. Dutchyn. Aspects in higher-order languages. *Science of Computer Programming*, 2006.

[4] This design depends on the MzScheme exception subsystem.