Ph.D. Thesis Proposal: Programming Language Support For Separation Of Concerns

Doug Orleans

May 22, 2002

Separation of Concerns (SOC)

"study in depth an aspect of one's subject matter in isolation, for the sake of its own consistency, all the time knowing that one is occupying oneself with only one of the aspects" [Edsger W. Dijkstra, 1976]

"divide and conquer" [ancient Roman motto]

Modularization

The decomposition of a problem into separate concerns should lead directly to the modularization of a program that solves the problem.

In other words: there should be a one-to-one mapping from problem concerns to program modules.

Programming Language Support for SOC

Object-oriented programming languages make it easy to modularize data-structure concerns (as classes), but functional concerns can end up scattered across multiple classes.

Aspect-oriented programming languages support the modularization of crosscutting concerns:

- AspectJ: aspects with pointcuts and advice
- HyperJ: hyperslices and hypermodules
- ComposeJ: composition filters
- DemeterJ: adaptive visitors

Fred

Fred is a new programming language that unifies OOP and AOP support for SOC.

Fred consists of a simple core language, based on predicate dispatching [Ernst, Kaplan, & Chambers 1998], plus syntactic sugar to emulate the higher-level constructs of other languages.

My prototype implementation is embedded into MzScheme. Using Fred with units [Flatt & Felleisen 1998] allows reusable components of crosscutting behavior.

Extensible Decisions

In OOP, whenever a message is sent, a decision occurs (dynamic dispatch), but the branches of the decision are specified separately: methods that correspond to the message signature. New branches can be added to a decision by defining methods in a new class.

In Fred, the branches are first-class entities, not attached to classes like methods are. The condition governing when a branch should be followed can be an arbitrary predicate involving the context of the message send.

The Core of Fred (1/3)

The behavior of a Fred program is specified as a set of **messages** and **branches**:

- (`define-msg` *name*) makes a message and binds it to *name* in the current environment.
- (`define-branch` *condition body*) makes a plain branch and adds it to the global set of branches; *condition* and *body* are procedures of one argument, a decision point.
- (`define-around` *condition body*) makes an around branch.

The Core of Fred (2/3)

When a message is sent to a list of argument values, a **decision point** is created, encapsulating the context of the message send; the context can be extracted with these accessors:

- (`dp-msg` *dp*) returns the message that was sent.
- (`dp-args` *dp*) returns the argument values that the message was sent to.
- (`dp-previous` *dp*) returns the previous decision point on the stack at the time the message was sent.

The Core of Fred (3/3)

After the decision point is created, the most precedent applicable branch is selected and its body procedure is invoked.

- A branch with predicate p is applicable to a decision point dp if $(p\ dp)$ is true.
- A branch with predicate p_1 precedes a branch with predicate p_2 if p_1 implies p_2.
- An around branch always precedes a plain branch.

`(follow-next-branch)` will invoke the body procedure of the next most precedent applicable branch.

Computing Implication

Logical implication of unrestricted predicates is undecidable in general.

Fred analyzes condition predicates only as far as logical connectors (`and`, `or`, `not`), type tests (`is-a?`, `integer?`), and equality and inequality relations (`eq?`, `=`, `<=`). Other subexpressions are treated as incomparable atoms in the logical formula (except for structural equivalence, up to alpha renaming).

If two predicates are incomparable, a "message ambiguous" error is raised. This can be detected at branch-definition time.

OOP in Fred (1/2)

```
(define-class person () (fname lname))
(define-msg full-name)
(define-branch
  (lambda (dp) (and (eq? (dp-msg dp) full-name)
                    (= (length (dp-args dp)) 1)
                    (is-a? (car (dp-args dp)) person))
  (lambda (dp)
    (let ((this (car (dp-args dp))))
      (string-append (get-fname this) " "
                     (get-lname this))))))
```

OOP in Fred (2/2)

```
(define-class knight (person) ())
(define-branch
  (lambda (dp) (and (eq? (dp-msg dp) full-name)
                    (= (length (dp-args dp)) 1)
                    (is-a? (car (dp-args dp)) knight)))
  (lambda (dp)
    (string-append "Sir " (follow-next-branch))))
```

Sample output:

```
> (define gandalf (make knight "Ian" "McKellen"))
> (full-name gandalf)
"Sir Ian McKellen"
```

AOP in Fred: Logging Concern

```
(define-around
  (lambda (dp) (and (is-a? (car (dp-args dp)) person))
                    (not (in-full-name-cflow? dp)))
  (lambda (dp)
    (let ((this (car (dp-args dp)))
          (msg (dp-msg dp)))
      (printf "~a received message ~a.~n"
        (full-name this) (msg-name msg))
      (follow-next-branch))))

(define (in-full-name-cflow? dp)
  (let ((prev (dp-previous dp)))
    (and prev (or (eq? (dp-msg prev) full-name)
                  (in-full-name-cflow? prev)))))
```

Syntactic Sugar

```
(define-class person () (fname lname))
(define-method full-name ((person this))
  (string-append (get-fname this) " " (get-lname this)))

(define-class knight (person) ())
(define-method full-name ((knight this))
  (string-append "Sir " (follow-next-branch)))

(define-before (&& (args person ..)
                   (! (cflowbelow (call full-name))))
  (with-msg-and-args (msg this . rest)
    (printf "~a received message ~a.~n"
            (full-name this) (msg-name msg))))
```

AspectJ Comparison

```
class Person {
  String fname, lname;
  String fullName() { return fname + " " + lname; }
}
class Knight extends Person {
  String fullName() { return "Sir " + super.fullName(); }
}
aspect Logging {
  before(Person p): call(* *(..)) && target(p) &&
                    !cflowbelow(call(* fullName(..))) {
    System.out.println(p.fullName() + " received message " +
                       thisJoinPoint.getSignature());
  }
}
```

Reusable Aspect: Caching Concern

```
(define memoize
  (unit (import memoize? dp-key
                invalidate? dp-keys)
        (export)

    (define-field cached-value ?)
    (define (clear-cache! c)
      (set-cached-value! c #f))

    (define-around memoize?
      (lambda (dp)
        (let ((key (dp-key dp)))
          (unless (get-cached-value key)
            (set-cached-value! key (follow-next-branch)))
          (get-cached-value key))))

    (define-before invalidate?
      (lambda (dp)
        (for-each clear-cache! (dp-keys dp))))))
```

Adapting the Aspect

```
(define memoize?
  (&& (call check) (args Item)))
(define (dp-key dp) (car (dp-args dp)))

(define invalidate?
  (&& (call add-item!) (args Container Item))
(define (dp-keys dp)
  (let loop ((node (car (dp-args dp))))
    (if (not node)
        '()
        (cons node (loop (get-parent node))))))

(invoke-unit memoize memoize? dp-key invalidate? dp-keys)
```

AspectJ Comparison

```
abstract aspect Memoize {
  abstract pointcut memoize(Object key);
  abstract pointcut invalidate(List keys);

  Object Object.cachedValue;
  static void clearCache(Object c) { c.cachedValue = null; }

  Object around(Object key) : memoize(key) {
    if (key.cachedValue == null)
      key.cachedValue = proceed(key);
    return key.cachedValue;
  }

  void before(List keys) : invalidate(keys) {
    Iterator i = keys.iterator();
    while (i.hasNext()) clearCache(i.next());
  }
}
```

Related Work: Aspect SandBox

The Aspect SandBox (ASB) project [Kiczales, Dutchyn, et.al.] "provides a framework for building simple interpreters for AOP languages". It consists of a Scheme interpreter for BASE, a simple OO language, and several extensions modeling different AOP styles, including AJD, the dynamic join point model of AspectJ.

Fred, on the contrary, unifies both OOP and AOP into a single mechanism: branches plus a precedence relation.

Also, ASB (so far) has no notion of reusable aspects; since ASB is intended to model AspectJ fairly closely, it will probably only have the same "abstract aspect" model as AspectJ. Fred with units provides a more flexible model of reuse.

Research Plan

The main goal is to show that Fred's support for SOC is an improvement on existing languages.

- add syntactic sugar to Fred to emulate the constructs of the major AOP languages (2 months)
- write some medium-sized programs in Fred and other languages (4 months)
- formal semantics and relative expressiveness (1 month)
- efficient implementation (1 month)
- tool support (1 month)
- write dissertation (6 months): September 2003

Syntactic Sugar

Fred already has syntax emulating CLOS (`define-class`, `define-method`) and AspectJ (`call`, `args`, `cflow`).

- HyperJ: `define-hyperslice`
- ComposeJ: `define-filter`
- DemeterJ: `define-traversal`, `define-visitor`
- Aspectual collaborations, mixin layers, logic metaprogramming, etc.

Example-based Comparison

- caching "challenge problem" [Ovlinger et.al.]
- cords library with optimizations [AOSD 02]
- GUI solitaire puzzle: model/view/controller
- multi-user programming environment: synchronization, security, resource control, persistence
- other example programs from papers about AOP languages

Semantics and Expressiveness

[Felleisen 1990] defined a formal notion of relative expressiveness of programming languages, given formal definitions of the semantics of the languages, based on structure-preserving transformations and behavioral equivalence. This is finer-grained than the hierarchy of computability: it distinguishes between Turing-complete languages. This could be used to prove (or provide proof sketches for):

- AspectJ is more expressive than Java
- Fred is more expressive than Scheme and CLOS
- Fred is at least as expressive as AspectJ [HyperJ, ComposeJ, etc]

Efficient Implementation

[Chambers and Chen 1999] describe implementation techniques for efficient predicate dispatching, by computing a dispatch tree from a set of predicates. This can eliminate redundant tests and order the tests for minimum tree depth. These techniques should apply to Fred just as well, although the tree would have to be recomputed whenever a new branch is defined.

Restricting the language of condition predicates may lead to optimizations, but at the risk of reduced expressiveness. The medium-sized programs may show that the full expressiveness of Fred isn't needed in practice.

Tool Support

Understanding aspect-oriented programs can be difficult, because the code for one concern may be affected by the code at some other concern, and the links between modules are not always evident from reading the code.

A smart code browser tool can display these links directly, providing more structure than the flat text of the source code. There are some IDE plugins for browsing AspectJ code; a similar tool could be developed as an extension to DrScheme.

Other Research Questions

- Does static typing affect SOC? Fred is dynamically typed; most AOP languages are based on Java, which is statically typed.
- Abstraction enforcement: how to protect code from being "interrupted" by other branches? Restrict the scope of condition predicates?
- Is it okay to have a single global table of branches, or do they need to be scoped?
- How should branch precedence be customized? How much does it need to be?

Conclusion

Fred is a programming language that supports separation of concerns by allowing program modules (units of branches) to correspond to problem concerns.

Fred is simple to understand, implement, and prove things about, yet general enough to support SOC better than many other OOP and AOP languages.

(auxiliary slides follow)

Predicate Analysis

All predicates are converted to disjunctive normal form. Then the following rules are applied:

- $(X_1 \vee X_2 \vee \ldots \implies Y_1 \vee Y_2 \vee \ldots) \iff (\forall i.\exists j.X_i \implies Y_j)$
- $(X_1 \wedge X_2 \wedge \ldots \implies Y_1 \wedge Y_2 \wedge \ldots) \iff (\forall j.\exists i.X_i \implies Y_j)$
- $(\neg X \implies \neg Y) \iff (Y \implies X)$
- $(($`is-a?` X $C_1) \implies$ `(is-a?` X $C_2)) \iff$ `(subclass?` C_1 $C_2)$
- $(($`eq?` X $Y) \implies$ `(`$\ldots X \ldots$`))` $\impliedby$ `(`$\ldots Y \ldots$`)`

Reimplementing Fred

In order to be clear on the semantics of Fred, it may help to reimplement Fred as something other than a Scheme library.

- ML library: easier to compare to a typed AOP language.
- implement a parser and interpreter: similar to ASB
- use a MOP: CLOS or tiny-clos
- extend MultiJava or GUD

Compound Units (1/2)

```
(define backlink
  (unit (import child-add? dp-parent dp-child)
        (export get-parent)

    (define-field parent ?)

    (define-after child-add?
      (lambda (dp)
        (set-parent! (dp-child dp) (dp-parent dp))))))
```

Compound Units (2/2)

```
(define memoize-tree
  (compound-unit
    (import memoize? dp-node
            child-add? dp-parent dp-child)
    (link [B (backlink child-add? dp-parent dp-child)]
          [UB ((unit (import dp-parent get-parent)
                     (export dp-ancestors)
                (define (dp-ancestors dp)
                  (let loop ((node (dp-parent dp)))
                    (if (not node)
                        '()
                        (cons node (loop (get-parent node))))))
                ) dp-parent (B get-parent))]
          [M (memoize memoize? dp-node
                      child-add? (UB dp-ancestors))])
    (export)))
```