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

Doug Orleans

April 19, 2002

Abstract

Separation of concerns is a useful problem-solving technique. Ideally, each program module should embody one (and only one) concern of the problem that the program is solving. In practice, this correspondence is limited by the constructs available in the programming language; some kinds of concerns are easier or harder to separate in a given language, and crosscutting concerns are hard to separate in any currently popular language. Programming languages should provide support for separation of concerns for as many kinds of concerns as possible, and they can achieve this while remaining conceptually simple. I describe a language called Fred that attempts to meet these criteria, based on predicate dispatching, aspect-oriented programming, and units. I provide an extended example of some reusable components that separate crosscutting concerns.

1 Introduction

In 1976, Edsger W. Dijkstra described a technique for problem solving [5, page 211]: "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"; he dubbed this technique **separation of concerns**. It is not easy to decide how a problem should be decomposed into separate concerns, but realizing that a decomposition is needed is an important first step in solving the problem.

Ideally, the decomposition of a problem into separate concerns should lead directly to the modularization of a program that solves the problem—that is to say, each program module should embody one (and only one) concern of the problem. The more this is true, the easier the program is to be understood, modified, or reused; the less this is true, the more the program tends to be tangled, brittle, and overspecialized.

In practice, this correspondence is limited by the constructs available in the programming language. Object-oriented languages make it easy to modularize data-structure concerns (as classes), but functional concerns can end up scattered across multiple classes. Functional languages such as ML have roughly the opposite problem. In addition, there are concerns that cannot be properly modularized using the constructs of *any* currently popular programming language, because the concerns cut across the modularization that is supported by the existing constructs. This realization has led to the design of **aspect-oriented programming (AOP)** [15] languages that support the modularization of crosscutting concerns.

AOP language design is an active area of research; some of the more prominent languages under development are AspectJ [14], HyperJ [19], ComposeJ [20], DemeterJ [12], and the language of aspectual components [16]. They all address the problem of crosscutting concerns, by extending Java [11] with new constructs for expressing concerns that could not be expressed modularly in plain Java.

I have developed my own AOP language called Fred [17]. It is a dynamically typed language with a very simple core set of constructs together with some syntactic sugar for expressing behavior in a variety of styles. Its central feature is a dynamic dispatch mechanism that generalizes predicate dispatching [6] to allow behavior that is common to multiple operations to be expressed in a single method, called a **branch**. My prototype implementation is embedded in MzScheme [8] as a set of procedures and macros; branches can be combined with MzScheme's **units** [9] to form reusable components of crosscutting behavior.

The goal of Fred is to be simple yet general, supporting the widest variety of concern modularization: the basic constructs should be easy to understand operationally, but they should be able (with suitable syntactic sugar) to emulate the constructs of the other AOP languages, including aspects, hyperslices, composition filters, adaptive visitors, and aspectual collaborations. A secondary goal is to be efficiently implementable, as much as possible without adding too many constraints on the expressibility of the language.

In summary, my proposed thesis is that programming languages should provide support for separation of concerns for as many kinds of concerns as possible, and that they can achieve this while remaining conceptually simple. The remainder of this paper describes a programming language that attempts to meet these criteria. Section 2 introduces the core constructs of Fred, along with some of the syntactic sugar. Section 3 shows an example of a Fred program with reusable components. Section 4 outlines the further research required to complete the dissertation.

2 Fred: A Simple Yet General AOP Language

As mentioned in section 1, Fred is embedded in MzScheme. This means that a Fred program may use syntax and procedures from Scheme [13]; in particular `define` and `lambda` will be needed.

The main idea behind Fred is that of **extensible decisions**. In an OOP language with dynamic dispatch, every message send triggers a decision, namely which method to execute. Unlike an `if` or `switch` statement, however, the decision is extensible: new methods can be added incrementally, in subclasses or sibling classes, without modifying code. In single-dispatch OOP, the decision depends solely on the run-time type of the receiver; with multiple dispatch, the decision can also depend on the run-time types of the other arguments. With predicate dispatching, the decision can depend on any arbitrary expression involving the arguments. In AspectJ, crosscutting behavior can be specified as extensions to existing decisions, in the form of advice, but the decision condition is even more general than predicate dispatching: a pointcut expression can involve the message, or the execution context or history. Each join point represents a decision to be made about what code to execute. Fred unifies all of these ways of extending decisions into one general dynamic dispatch mechanism.

There are four basic kinds of behavioral entities in Fred: **messages**, **primary branches**, **around branches**, and **decision points**. Fred provides syntax for defining the first three:

- `(define-msg` ***`name`***`)` creates a message and binds it to the variable ***`name`*** in the current lexical environment. Messages are procedures and can be sent to a list of values using the same syntax as procedure application.
- `(define-branch` ***`condition body`***`)` and `(define-around` ***`condition body`***`)` create a primary or around branch whose condition predicate is the value of ***`condition`*** and whose body is the value of ***`body`***. Both arguments must evaluate to procedures of one argument, a decision point. The branch is added to the global set of branches.

Decision points are not created explicitly; when a message is sent, a decision point is implicitly created and passed to the condition predicates of every defined branch. The set of branches whose condition predicates return true (that is, any value besides `#f`) for a given decision point is known as the set of **applicable branches**; it is a runtime error ("message not understood") if this set is empty. The most precedent branch from this set is selected and the decision point is passed to its body procedure; it is a runtime error ("message ambiguous") if there is more than one most precedent applicable branch, unless they are all around branches, in which case one is selected arbitrarily. The branch body procedure can call the special function `follow-next-branch` to pass the decision point to the body procedure of the most precedent applicable branch that is not already being executed. The precedence relation between branches is based on logical implication of condition predicate expressions, except that around branches always precede primary branches; this relation is described in further detail later in this section.

There are three procedures for inspecting the context of a decision point:

- `(dp-msg` *`dp`*`)` returns the message that was sent. (`(msg-name` *`msg`*`)` returns the name of a message as a symbol.)
- `(dp-args` *`dp`*`)` returns the list of values the message was sent to.
- `(dp-previous` *`dp`*`)` returns the decision point that was being processed when the message was sent.

These basic building blocks are enough to define the behavior of a program separated by concern. For example, using MzScheme's structure type facility, we can make a simple `person` class:

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

A `person` object has two fields, `fname` and `lname`. When the `full-name` message is sent to a `person` object, the values of the `fname` and `lname` fields of that object are concatenated (with a space in between). Thus `full-name` acts as a method on the `person` class, although it is defined outside of the class definition and could appear in a separate part of the program.

We can also make a `knight` subclass with new behavior for handling `full-name`:

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

Notice that this branch's condition predicate overlaps with that of the previous one: if `full-name` is sent to a `knight` object, then both conditions will be true. In this case, the second branch has precedence, because its condition predicate logically implies the first branch's—that is, the first branch's condition predicate is always true whenever the second branch's condition predicate is true. In particular, `(person? x)` is always true whenever `(knight? x)` is true, because `knight` is a subclass of `person`.

The idea of branch precedence being based on logical implication between condition predicates is taken directly from Ernst et. al.'s predicate dispatching system [6]. The main advantage of this is that branch precedence is a straightforward generalization of the usual method overriding rules in object-oriented languages: subclass methods override superclass methods, or in the case of multiple dispatch, more specific methods override less specific methods.

However, sometimes you want a branch with a very general condition predicate to precede other branches with more specific condition predicates. For example, suppose you wanted to trace all messages sent to `person` objects:

```
(define-around
  (lambda (dp)
    (person? (car (dp-args dp))))
  (lambda (dp)
    (printf "Received message ~a.~n" (msg-name (dp-msg dp)))
    (follow-next-branch)))
```

Its condition predicate does not logically imply any of the condition predicates of the other branches, but it needs to be followed before they do. (In particular, it needs to precede the first branch above, since that branch's body procedure does not call `follow-next-branch`!) Making it an around branch causes it to precede any primary branch, so it will be followed before the others.

Logical implication of predicates is undecidable in general, so Fred can only have a conservative approximation to the implication relation. To determine predicate implication, Fred uses: the subtype relation, both for user-defined structure types and built-in Scheme types (such as the numeric type hierarchy); substitution of equals for equals (e.g. `(eq? x 'foo)` implies `(symbol? x)` because `(symbol? 'foo)` is true); and propositional calculus rules (such as `(and` P Q`)` implies P). Although Fred currently computes this relation between applicable branches at message send time, the relation could instead be computed between all branches at branch definition time. A static analyzer could even guarantee that "message ambiguous" errors will never occur, by rejecting a program unless it can determine that all pairs of branch predicates are either related by implication in one direction or logically exclusive, i.e. one implies the negation of the other.

Fred provides some helper functions and syntax to make condition predicates more concise, in the manner of AspectJ's pointcut designators:

- `(call` *`msg`*`)` produces a condition predicate that returns true for any decision point whose message is *`msg`*.
- `(args` *`type-pred`* ... [`..`]`)` produces a condition predicate that returns true for any decision point such that each *`type-pred`* returns true for the corresponding argument of the decision point. The predefined type predicate `?` returns true for all values. If `..` is specified at the end of the `args` form, then the number of arguments may be more than the number of type predicates.
- `(cflow` *`condition`*`)` produces a condition predicate that returns true if *`condition`* is true for the decision point or any of its predecessors (using `dp-previous` to walk up the stack). `(cflowbelow` *`condition`*`)` is the similar but does not check the decision point itself. (An example using `cflow` to solve a "vanishing aspects" problem appears in [17].)
- `(&&` *`condition`* ...`)`, `(||` *`condition`* ...`)`, and `(!` *`condition`*`)` can be used to combine condition predicates into logical formulas.

Fred also provides some syntax to bind some variables in the body of a branch to parts of the decision point, similar to arguments to advice in AspectJ:

- `(with-msg (`*`msg`*`)` *`body`* `...)` produces a body procedure that binds *`msg`* to the message of the decision point in the scope of the *`body`* expressions.

- `(with-args` *`formals body`* `...)` produces a body procedure that binds *`formals`* to the arguments of the decision point in the scope of the *`body`* expressions.

- `(with-msg-and-args` *`formals body`* `...)` produces a body procedure that binds `formals` to the message and arguments of the decision point in the scope of the *`body`* expressions.

Thus, the three example branches could be written as follows:

```
(define-branch (&& (call full-name) (args person?))
  (with-args (this)
    (string-append (person-fname this) " "
                   (person-lname this)))))
(define-branch (&& (call full-name) (args knight?))
  (lambda (dp)
    (string-append "Sir " (follow-next-branch))))
(define-around (args person? ..)
  (with-msg (msg)
    (printf "Received message ~a.~n" (msg-name msg))
    (follow-next-branch)))
```

Fred also provides syntactic sugar for concise definition of branches that act like ordinary predicate dispatching methods, i.e. their condition predicate only involves a single `call` test and some predicates over the arguments:

- `(define-method` *`msg formals`* [`&` *`pred`*] *`body`* `... )` adds a primary branch whose condition predicate tests that the decision point message is *`msg`* and that *`pred`* is true (if supplied), with the arguments bound to *`formals`*. The branch body also has the arguments bound to *`formals`*.

This allows us to rewrite the first two example branches as follows:

```
(define-method full-name (this) & (person? this)
  (string-append ((person-fname this) " " (person-lname this))))
(define-method full-name (this) & (knight? this)
  (string-append "Sir " (follow-next-branch)))
```

Additionally, each formal parameter in the *`formals`* list can be replaced by an application whose first argument is a variable, which is bound to the corresponding argument:

```
(define-method full-name ((person? this))
  (string-append ((person-fname this) " " (person-lname this))))
(define-method full-name ((knight? this))
  (string-append "Sir " (follow-next-branch)))
```

Two abbreviations also exist for around branches:

- `(define-before` *`condition body`*`)` adds an around branch whose body invokes the *`body`* procedure and then follows the next branch.

- `(define-after` *`condition body`*`)` adds an around branch whose body follows the next branch and then invokes the *`body`* procedure.

Thus we can rewrite the third example branch as follows:

```
(define-before (args person? ..)
  (with-msg (msg)
    (printf "Received message ~a.~n" (msg-name msg))))
```

Instead of using MzScheme's structure mechanism to define data structures, Fred provides some syntax for defining classes and fields:

- `(define-class` *`name parent`* `...)` creates a class with the given `parent` classes and binds it to the variable *`name`* in the current lexical environment.
- `(define-field` *`name type`* [*`default`*]`)` creates storage for the given *`type`*, which is either a class or a type predicate. It defines two messages in the current lexical environment, `get-`*`name`* and `set-`*`name`*`!`, and adds two branches to get and put values from and to the storage. The `get-`*`name`* branch returns the value of *`default`* if no value has been stored for its argument using the `set-`*`name`*`!` branch. `(add-field` *`name type`* [*`default`*]`)` adds the two branches, assuming the two messages have already been defined.

Fred also provides two messages for instantiating classes, and one procedure for testing instances:

- `(make` *`class initarg`* `...)` creates an instance of *`class`* and sends the `init` message to the instance and the *`initarg`* values. The default branch for `init` does nothing.
- `(is-a?` *`obj class`*`)` returns true if *`obj`* is an instance of *`class`* or one of its descendants.

Also, a class can be used in place of a type predicate in an `args` or `define-method` form. Thus we can rewrite the previous example as follows:

```
(define-class person)
(define-field fname person)
(define-field lname person)
(define-method init ((person this) fname lname)
  (set-fname! this fname)
  (set-lname! this lname))
(define-msg full-name)
(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 ..)
  (with-msg (msg)
    (printf "Received message ~a.~n" (msg-name msg))))
```

3 An Example

In order to give a better feel for how the mechanisms described in section 2 fit together to make reusable components of crosscutting concerns, I will present a complete example of a crosscutting component used with two different base programs. The component is itself built up from several smaller reusable components. The example is based on the "challenge problem" of Ovlinger et. al. [18].

The example uses MzScheme's **units** facility [9]. There are two kinds of units, **atomic** and **compound**:

- `(unit (import` *`iname`* `...) (export` *`ename`* `...)` *`body`* `...)` creates an atomic unit. The *`iname`* variables are bound in the scope of the *`body`* expressions, which must define the `ename` variables.

- `(compound-unit (import` *`iname`* `...) (link (`*`tag`* `(`*`unit linkage`* `...)) ...) (export (`*`tag ename`*`) ...))` creates a compound unit from other units. The units are linked together by assigning each unit a unique *`tag`* variable, and providing *`linkage`* specifications for each of a unit's imports. A *`linkage`* specifications can either be one of the *`iname`* variables or the form `(`*`tag var`*`)` where *`var`* is one of the export variables from the unit corresponding to *`tag`*. Similarly, the *`ename`* variables come from units corresponding to their *`tag`* variables.

Units are invoked with `(invoke-unit` *`unit arg`* `...)`. Invoking an atomic unit binds its import variables to the values of the `arg` expressions, then evaluates its body expressions in order. Invoking a compound unit also binds its import variables, but then invokes each of its sub-units in order, binding their import variables to the exported values from the other units.

The first base program is shown in figure 1. It defines an abstract `Item` class and two concrete subclasses, `Simple` and `Container`. `Simple` objects have weight, while `Container`s have capacity and a list of contained `Items`. The `check` operation determines whether a `Container`, or any `Containers` inside it, are over capacity, and prints a message if so. Figure 2 defines a test case for the program.

Figure 1: The container unit

```
(define container
  (unit (import)
        (export Item get-name
                Simple get-weight
                Container get-capacity get-contents add-item!
                check)

    (define-class Item)
    (define-field name Item)
    (define-method init ((Item this) name)
      (set-name! this name))

    (define-class Simple Item)
    (define-field weight Simple)
    (define-method init ((Simple this) name weight)
      (init this name)
      (set-weight! this weight))

    (define-class Container Item)
    (define-field capacity Container)
    (define-field contents Container '())
    (define-method init ((Container this) name capacity)
      (init this name)
      (set-capacity! this capacity))
    (define-msg add-item!)
    (define-method add-item! ((Container this) (Item i))
      (set-contents! this (append (get-contents this) (list i))))

    (define-msg check)
    (define-method check ((Simple this))
      (printf "Simple object ~a weighs ~a~n" (get-name this) (get-weight this))
      (get-weight this))
    (define-method check ((Container this))
      (let ((total (apply + (map check (get-contents this)))))
        (printf "Container ~a weighs ~a~n" (get-name this) total)
        (when (> total (get-capacity this))
          (printf "Container ~a overloaded~n" (get-name this)))
        total))

    ))
```

Figure 2: Testing the `container` unit

```
(define test-container
  (unit (import Simple Container add-item! check)
        (export main)
    (define (main)
      (let ((c1 (make Container "basket" 4))
            (c2 (make Container "bowl" 1))
            (c3 (make Container "bag" 1))
            (apple (make Simple "apple" 1))
            (pencil (make Simple "pencil" 1))
            (orange (make Simple "orange" 1))
            (kiwi (make Simple "kiwi" 1))
            (banana (make Simple "banana" 1)))
        (add-item! c3 kiwi)
        (add-item! c2 c3)
        (add-item! c2 apple)
        (add-item! c1 orange)
        (add-item! c1 pencil)
        (add-item! c1 c2)
        (check c1)
        (add-item! c2 banana)
        (check c1)
        (void)))
  ))

(define container-program
  (compound-unit
    (import)
    (link [C (container)]
          [TC (test-container (C Simple) (C Container)
                              (C add-item!) (C check))])
    (export (TC main))))
```

Figure 3: The `memoize` unit

```
;; Memoize a computation: the first time a decision point occurs,
;; store the return value, and return it the next time a decision
;; point with the same key occurs instead of recomputing.
(define memoize
  (unit (import memoize? dp-key
                invalidate? dp-keys)
        (export)

    (define-field cached-value ?)
    (define (clear-cache! c)
      (printf "clear cache~n")
      (set-cached-value! c #f))

    (define-around memoize?
      (lambda (dp)
        (let ((key (dp-key dp)))
          (if (not (get-cached-value key))
              (set-cached-value! key (follow-next-branch))
              (printf "using cached value~n"))
          (get-cached-value key))))

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

Notice that the `check` operation does a recursive traversal over a composite `Container` object. An optimization would be to memoize the results for each container and sub-container, but the caches would have to be invalidated when a container's contents are modified (with `add-item!`). Since memoization is a common technique that is applicable in many different situations, we can make a generic `memoize` unit (figure 3) that is parameterized over two condition predicates: `memoize?`, which returns true for all decision points corresponding to invocations of the operation that we want to memoize, and `invalidate?`, which returns true for all decision points corresponding to operations that cause some caches to be invalidated. The `memoize` unit also imports two functions that extract relevant data from a decision point that matches one of the two conditions: `dp-key` extracts the input to the function being memoized, while `dp-keys` extracts the list of keys whose caches should be invalidated. The cache is implemented as a field, since fields can be defined outside of any particular class; the type predicate for the field is `?`, since the type of the keys is irrelevant.

Notice that the `memoize` unit does not export anything, it simply imports some condition predicates and defines some branches that use them. Branches are not bound to variables; they exist in a global table, so there's nothing to export. This is in contrast to Findler and Flatt's style of programming with units and mixins [7], where each unit imports some classes, defines classes that extend the imported classes, and exports the new classes. The effect of such a unit is to add behavior to a class by replacing it with a subclass that has the added behavior. In Fred, however, behavior can be added directly by defining branches, so there's no need to export replacement classes. This approach is simpler and potentially more efficient, since there isn't an artifical extension to the inheritance chain with every increment.

Now in order to implement the `dp-keys` function required by the `memoize` unit, we need to

Figure 4: The `backlink` unit

```
;; Keep track of the parent node for each node in a tree structure.
;; child-add? is a condition predicate representing an addition of a
;; new child node to a parent node.  dp-parent and dp-child extract
;; the parent and child from the addition dp.  get-parent returns the
;; parent node of a node, or #f if it is a root.
(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))))))
```

keep track of what containers an item is inside; we only know what items are inside a container, but there is no link pointing in the other direction. Keeping backlinks for a tree structure is another common technique that we can implement using a generic `backlink` unit (figure 4). This unit imports one condition predicate, `child-add?`, which returns true for all decision points corresponding to the addition of a new child node to a parent node; the `dp-parent` and `dp-child` functions extract the parent and child nodes from the decision point. The `backlink` unit exports one message, `get-parent`, which returns the parent node of a given node, or `#f` if it is a root node (i.e. it hasn't been added to any parent). Similar to the `memoize` unit, it stores the backlink in a field that is not attached to any class, since we don't care what the type of nodes is.

Given these two generic units, we can build a compound unit that memoizes a recursive function over a tree structure. The `memoize-tree` unit (figure 5) is more specialized than the `memoize` unit, but still generic enough to be used with any kind of tree structure data types. The `dp-keys` function is provided to the `memoize` unit by an adapter unit (defined inline) that traverses the backlinks from a node to the root of the tree it belongs to, using the `get-parent` function exported from the `backlink` unit. The `memoize?` and `child-add?` condition predicates (plus their extractors) are imported and passed on to the sub-units.

We can further specialize this unit for a tree structure that is implemented by a set of classes using the Composite design pattern [10]. The `memoize-composite-function` unit (figure 6) imports two classes, `Component` and a subclass `Composite`, as well as two messages, `func` and `add-component!`. The former is the function to be memoized, and takes a `Component` object as its first argument; the latter takes at least two arguments, a `Composite` object C and a `Component` object O, and adds O to C. These are adapted to the `memoize-tree` unit by providing condition predicates and extractors in terms of the imported classes and messages.

We now have two different generic crosscutting components that memoize a recursive function over a tree structure: `memoize-tree` is similar to the AspectJ implementation of the challenge problem in [18], which uses abstract pointcuts in the style of Clarke and Walker [3], while `memoize-composite-function` is similar to the implementation using aspectual collaborations, which imports a participant graph of classes. The latter is simpler to use when the base program does actually use the Composite design pattern; it's just a matter of name mapping (figure 7). (Figure 8 shows the program output comparing the effects of using and not using the caching aspect.) However, the former can be used in a wider variety of situations, such as an implementation using Scheme lists and `set-cdr!`. Fred supports both styles of parameterized components equally well, due to the generality of units and branches.

Figure 5: The `memoize-tree` unit

```
;; Memoize a computation on nodes of a tree that recurs on the node's
;; children.  Invalidate a node's cache whenever a new child is about
;; to be added to the node.
(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)))
```

Figure 6: The `memoize-composite-function` unit

```
;; Memoize a recursive function over a set of classes that use the
;; Composite pattern.  Invalidate the cache when a component is added
;; to a composite.
(define memoize-composite-function
  (compound-unit
    (import Component func Composite add-component!)
    (link [A ((unit (import Component func Composite add-component!)
                    (export func-call? dp-node
                            child-add? dp-parent dp-child)

                (define func-call?
                  (&& (call func) (args Component ..)))
                (define (dp-node dp)
                  (car (dp-args dp)))           ; the Component

                (define child-add?
                  (&& (call add-component!) (args Composite Component ..)))
                (define (dp-parent dp)
                  (car (dp-args dp)))           ; the Composite
                (define (dp-child dp)
                  (cadr (dp-args dp)))          ; the Component
                ) Component func Composite add-component!)]
          [M (memoize-tree (A func-call?) (A dp-node)
                           (A child-add?) (A dp-parent) (A dp-child))])
    (export)))
```

Figure 7: Putting it all together

```
(define cached-container-program
  (compound-unit
    (import)
    (link [C (container)]
          [M (memoize-composite-function (C Item) (C check)
                                         (C Container) (C add-item!))]
          [TC (test-container (C Simple) (C Container)
                              (C add-item!) (C check))])
    (export (TC main))))

(define caching-comparison
  (unit (import main cached-main)
        (export)
    (printf "Running the program without caching:~n")
    (main)
    (printf "~nRunning the program with caching:~n")
    (cached-main)
    (newline)
    ))

(define (compare base cached)
  (invoke-unit
   (compound-unit
     (import)
     (link [B (base)]
           [C (cached)]
           [P (caching-comparison (B main) (C main))])
    (export))))

(compare container-program cached-container-program)
```

In order to demonstrate that the `memoize-composite-function` unit is truly reusable, I have constructed a completely different base program that also uses the Composite design pattern and has a recursive function that could be optimized with memoization. Figure 9 shows a `GUI` unit that computes an `Image` from a tree of nested `Widget` objects; figure 10 shows a test program. Again, using the aspect is just a matter of name mapping (figure 11). The comparison output is shown in figure 12.

4 Research Plan

Perhaps the most important work to be done to complete the dissertation is to compare the capabilities of Fred with the major AOP languages. I believe that it can do most of the interesting things that they can do, and with some additions to the core facilities it can do the rest. The first step in the comparison will be to define syntactic sugar for the features of the other languages; this has been done for the basic features of AspectJ, namely the primitive pointcut designators `call`, `args` (which subsumes `target`), and `cflow`. A similar approach should be suitable for defining aspects (both concrete and abstract), as well as HyperJ's hyperslices, ComposeJ's composition filters, DemeterJ's adaptive visitors, and aspectual collaborations. My hunch is that all of these constructs can be defined as units containing branches.

The next step will be to come up with a good set of examples that can be defined in all the different languages, and show that the essential features of the implementations are preserved when translated to Fred. The caching example presented in this paper is one candidate; the cords library from my first paper about Fred [17] is another. The papers describing the other languages should also provide plenty of examples to choose from.

One major difference between Fred and the other languages is that Fred is dynamically typed while they are all statically typed, being extensions to Java. I don't think this will prohibit useful comparisons between the languages; the type declarations can simply be erased from the programs in the other languages, and the crosscutting nature of the concerns implemented by the programs will still be evident. That is to say, the support for separation of concerns in AOP languages is not provided by the static type system, but by the features of the dynamically typed subset. If this turns out not to be true for some of the languages, then I may need to add some kind of static type system to Fred in order to make a fair comparison.

A further interesting exercise in language comparison would be to formally define the dynamic semantics of (some interesting subset of) the languages, and use Felleisen's notion of **expressibility** to demonstrate that Fred is at least as expressible as the other AOP languages, if not strictly more expressible. While I may not have the time to complete formal proofs, I should be able to come up with example programs that intuitively satisfy the requirements for comparing expressbility.

There are a number of capabilities that I have already been planning to add to Fred in order to emulate the features of other AOP languages. One is the ability to declare inheritance relations between classes separately from the definitions of the classes, similar to the `declare parents` feature of AspectJ. (This feature is also present in BeCecil [2], a spiritual ancestor to Fred.)

Another feature that would be useful is to be able to bind variables in a branch's condition predicate that are visible to the branch's body procedure. The predicate dispatching language of Ernst et. al. [6] has this feature, which is very similar to the context exposure feature of AspectJ: pointcuts can bind variables that can be accessed in advice. The example program in section 3 shows an alternate mechanism that does something similar: selectors that extract information from a decision point can be supplied to a unit alongside a condition predicate.

It may turn out that a finer-grained mechanism for overriding the precedence relation is needed than simply having primary branches and around branches. AspectJ has the `dominates` relation between aspects that contributes to advice precedence; something along these lines could be added to Fred.

There is much room for improvement in the efficiency of Fred's current implementation. Chambers and Chen [1] describe several techniques for efficiently implementing predicate dispatching, by computing a dispatch tree from a set of predicates. These techniques can be used in Fred, although the dispatch tree would have to be recomputed when a new branch is added; there may be some way to structure the tree so that it would only need partial recomputation. There may also be some static analysis techniques that could assist in optimization.

One common criticism of AOP is that it seems to break one of the benefits of encapsulation: the code for one concern may be affected by the code in some other concern, but you can't tell that from looking at one concern in isolation. This problem could be alleviated with a smart code browser tool that would show exactly what concerns have code that could affect the code you're looking at. The AspectJ team has developed aspect browser plugins for various development environments; something similar could be developed as an extension to DrScheme [4].

References

[1] C. Chambers and W. Chen. Efficient multiple and predicate dispatching. In *Proceedings of OOPSLA '99*, Denver, CO, November 1999.

[2] C. Chambers and G. Leavens. Bececil, a core object-oriented language with block structure and multimethods: Semantics and typing. In *Proceedings of the The Fourth International Workshop on Foundations of Object-Oriented Languages (FOOL 4)*, Paris, France, January 1997.

[3] S. Clarke and R. J. Walker. Separating crosscutting concerns across the lifecycle: From composition patterns to AspectJ and Hyper/J. Technical Report UBC-CS-2001-05, University of British Columbia, Vancouver, Canada, 2001.

[4] J. Clements, P. T. Graunke, S. Krishnamurthi, and M. Felleisen. Little languages and their programming environments. Monterey Workshop, 2001.

[5] E. W. Dijsktra. *A Discipline of Programming.* Prentice-Hall, 1976.

[6] M. D. Ernst, C. Kaplan, and C. Chambers. Predicate dispatching: A unified theory of dispatch. In *Proceedings of ECOOP '98, the 12th European Conference on Object-Oriented Programming*, pages 186–211, Brussels, Belgium, July 20–24 1998.

[7] R. B. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of ICFP*, 1998.

[8] M. Flatt. PLT MzScheme: Language manual. Technical Report TR97-280, Rice University, 1997.

[9] M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 236–248, 1998.

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

[11] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison Wesley, 1996.

[12] G. Hulten, K. Lieberherr, J. Marshall, D. Orleans, and B. Samuel. *DemeterJ User Manual.* `http://www.ccs.neu.edu/research/demeter/`.

[13] R. Kelsey, W. Clinger, and J. Rees. Revised^5 report on the algorithmic language scheme, February 1998.

[14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming*, 2001.

[15] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, June 1997.

[16] K. Lieberherr, D. H. Lorenz, and J. Ovlinger. Aspectual collaborations for collaboration-oriented concerns. Technical Report NU-CCS-01-08, College of Computer Science, Northeastern University, Boston, MA 02115, Nov. 2001.

[17] D. Orleans. Incremental programming with extensible decisions. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD)*, Enschede, The Netherlands, April 2002.

[18] J. Ovlinger, K. Lieberherr, and D. Lorenz. Aspects and modules combined. Technical Report NU-CCS-02-03, College of Computer Science, Northeastern University, Boston, MA, March 2002.

[19] P. Tarr and H. Ossher. *Hyper/J User and Installation Manual*. IBM T. J. Watson Research Center, Yorktown Heights, NY, USA, 2000.

[20] J. Wichman. ComposeJ: The development of a preprocessor to facilitate composition filters in the Java language. MSc. thesis, Dept. of Computer Science, University of Twente, Enschede, the Netherlands, December 1999.

A Auxiliary Figures

Figure 8: Output of `container` comparison

```
Running the program without caching:
Simple object orange weighs 1
Simple object pencil weighs 1
Simple object kiwi weighs 1
Container bag weighs 1
Simple object apple weighs 1
Container bowl weighs 2
Container bowl overloaded
Container basket weighs 4
Simple object orange weighs 1
Simple object pencil weighs 1
Simple object kiwi weighs 1
Container bag weighs 1
Simple object apple weighs 1
Simple object banana weighs 1
Container bowl weighs 3
Container bowl overloaded
Container basket weighs 5
Container basket overloaded

Running the program with caching:
clear cache
clear cache
clear cache
clear cache
clear cache
clear cache
Simple object orange weighs 1
Simple object pencil weighs 1
Simple object kiwi weighs 1
Container bag weighs 1
Simple object apple weighs 1
Container bowl weighs 2
Container bowl overloaded
Container basket weighs 4
clear cache
clear cache
using cached value
using cached value
using cached value
using cached value
Simple object banana weighs 1
Container bowl weighs 3
Container bowl overloaded
Container basket weighs 5
Container basket overloaded
```

Figure 9: The GUI unit

```
(define GUI
  (unit (import)
        (export Image show overlay translate
                Widget get-image
                Label get-text
                Canvas
                Panel add-widget!)

    (define-class Image)
    (define-field bits Image '())
    (define-method init ((Image this) bits) (set-bits! this bits))
    (define-msg show)
    (define-method show ((Image it)) (reverse (get-bits it)))
    (define-msg overlay)
    (define-method overlay ((Image i1) (Image i2))
      (make Image (append (get-bits i1) (get-bits i2))))
    (define-msg translate)
    (define-method translate ((Image this) (complex? vector))
      (make Image (map (lambda (bit) (cons (+ vector (car bit)) (cdr bit)))
                       (get-bits this))))

    (define-class Widget)

    (define-class Label Widget)
    (define-field text Label)
    (define-method init ((Label this) text) (set-text! this text))

    (define-class Canvas Widget)
    (define-msg set-image!)
    (add-field image Canvas)
    (define-method init ((Canvas this) image) (set-image! this image))

    (define-class Panel Widget)
    (define-field widgets Panel '())
    (define-msg add-widget!)
    (define-method add-widget! ((Panel this) (Widget w) (complex? loc))
      (set-widgets! this (cons (cons loc w) (get-widgets this)))
      (void))

    (define-msg get-image)
    (define-method get-image ((Label this))
      (make Image (list (list 0 (get-text this)))))
    (define-method get-image ((Panel this))
      (let loop ((widgets (get-widgets this)))
        (if (null? widgets)
            (make Image '())
            (overlay (translate (get-image (cdar widgets)) (caar widgets))
                     (loop (cdr widgets)))))))
```

Figure 10: Testing the GUI unit

```
(define test-GUI
  (unit (import Image show
                Label
                Canvas
                Panel add-widget! get-image)
        (export main)
    (define (main)
      (let ((main-panel (make Panel))
            (text-panel (make Panel))
            (picture-panel (make Panel)))
        (add-widget! text-panel (make Label "hello") 0)
        (add-widget! text-panel (make Label "world") 6)
        (add-widget! main-panel text-panel 1+1i)
        (add-widget! picture-panel
                     (make Canvas (make Image '((0 x) (1+1i x) (2+2i x))))
                     0)
        (add-widget! picture-panel
                     (make Canvas (make Image '((2 x) (1+1i x) (0+2i x))))
                     0)
        (add-widget! main-panel picture-panel 5+2i)
        (add-widget! main-panel (make Label "OK") 5+6i)
        (printf "~s~n" (show (get-image main-panel)))
        (add-widget! text-panel (make Label "!") 11)
        (printf "~s~n" (show (get-image main-panel)))
        (void)))))

(define GUI-program
  (compound-unit
    (import)
    (link [G (GUI)]
          [TG (test-GUI (G Image) (G show)
                        (G Label)
                        (G Canvas)
                        (G Panel) (G add-widget!) (G get-image))])
    (export (TG main))))
```

Figure 11: Reusing the `memoize-composite-function` unit

```
(define cached-GUI-program
  (compound-unit
    (import)
    (link [G (GUI)]
          [M (memoize-composite-function (G Widget) (G get-image)
                                         (G Panel) (G add-widget!))]
          [TG (test-GUI (G Image) (G show)
                        (G Label)
                        (G Canvas)
                        (G Panel) (G add-widget!) (G get-image))])
    (export (TG main))))

(compare GUI-program cached-GUI-program)
```

Figure 12: Output of `GUI` comparison

```
Running the program without caching:
((1+1i "hello") (7+1i "world") (7+4i x) (6+3i x) (5+2i x) (5+4i x)
 (6+3i x) (7+2i x) (5+6i "OK"))
((1+1i "hello") (7+1i "world") (12+1i "!") (7+4i x) (6+3i x) (5+2i x)
 (5+4i x) (6+3i x) (7+2i x) (5+6i "OK"))

Running the program with caching:
clear cache
clear cache
clear cache
clear cache
clear cache
clear cache
clear cache
((1+1i "hello") (7+1i "world") (7+4i x) (6+3i x) (5+2i x) (5+4i x)
 (6+3i x) (7+2i x) (5+6i "OK"))
clear cache
clear cache
using cached value
using cached value
using cached value
using cached value
((1+1i "hello") (7+1i "world") (12+1i "!") (7+4i x) (6+3i x) (5+2i x)
 (5+4i x) (6+3i x) (7+2i x) (5+6i "OK"))
```