Incremental Programming with Extensible Decisions

Doug Orleans
College of Computer Science
Northeastern University
360 Huntington Avenue 161CN
Boston, Massachusetts 02115 USA
dougo@ccs.neu.edu

ABSTRACT

Languages that support incremental programming, that is, the construction of new program components by specifying how they differ from existing components, allow for clean separation of concerns. Object-oriented languages support incremental programming with inheritance and dynamic dispatch features: whenever a message is sent, a decision occurs, but the branches of the decision can be specified in separate components. Aspect-oriented programming and predicate dispatching both introduce language mechanisms that improve on this support by allowing an extensible decision to depend on information about the message send other than just the dynamic type of the receiver or arguments. A small prototype language is presented that unifies the best features of these mechanisms, providing uniform support for incremental programming whether concerns are crosscutting or not. The language is demonstrated with a running example, a small data structure library that is incrementally extended with optimizations and new operations.

Keywords

Aspect-oriented programming, predicate dispatching, incremental programming, extensibility, separation of concerns

1. INTRODUCTION

Incremental programming is defined by Cook and Palsberg as the construction of new program components by specifying how they differ from existing components [9]. A language that supports incremental programming allows for clean separation of concerns, because a component that involves multiple concerns can be expressed as a sequence of components, one per concern, each one extending or overriding the behavior in the previous components without requiring modification or duplication of code. This can improve the understanding, maintenance, and re-use of software.

Object-oriented programming (OOP) languages typically support incremental programming with inheritance and dynamic dispatch features. Whenever a message is sent, a decision occurs, but the branches of the decision can be specified in separate components, as methods whose signature matches the message. The decision of what branch to follow depends on the dynamic type of the receiver (or on the dynamic types of all of the arguments of the message send, in a language with multiple dispatch such as CLOS [29], Dylan [27], or MultiJava [8]). New branches can be added to this decision by defining methods in (or specialized on) a new class; when a message is sent to an instance of the new class, the new method will be invoked. The new method can provide a new alternate choice for the decision, or it can extend or override the behavior of an existing method on an ancestor class.

A significant restriction of this mechanism of extensible decisions in OOP languages is that different methods corresponding to the same message must be attached to (or specialized on) different classes; only type-dependent decisions can be made extensible, and the decision must be based on the type of the receiver (or the arguments). Many behavioral design patterns [14] are essentially workarounds for this restriction. The State pattern, for example, is a way to implement state-dependent dispatch using OOP's type-dependent dispatch, by making one class per state and having objects delegate state-dependent messages to their state objects. The behavior for new states can then be added incrementally by making new state classes. However, the delegation has a runtime cost and can be a coding burden; moreover, the state objects must be manually kept up to date when the state condition changes, so the states can't be implicitly determined by other variables that may independently change.

A different solution for this kind of problem is to extend the programming language to make it easier to express programs in a way that matches the design, rather than to change a program's design to fit the mechanisms of the language. **Aspect-oriented programming (AOP)** [20] and **predicate dispatching (PD)** [11] both introduce language mechanisms that allow an extensible decision to depend on information about the message send other than just the dynamic type of the receiver or arguments.

Aspect-oriented programming languages enable separation of crosscutting concerns. In AspectJ [19], a general-purpose AOP language that extends Java [15], crosscutting behavior is defined as **advice**, where each piece of advice has a **point-**

To appear in the 1st International Conference on Aspect-Oriented Software Development (AOSD), April 2002, Enschede, The Netherlands.

cut which specifies when the advice is applicable. A pointcut describes a set of **join points**, points in the execution of a running program where behavior can be attached—in other words, places where extensible decisions occur, including message sends. A pointcut can be an arbitrary boolean expression involving information available about the join point.

Predicate dispatching is a form of dynamic dispatch that unifies and extends the dispatch mechanisms found in many programming languages, including object-oriented single and multiple dispatch. In the PD language described by Ernst, Kaplan, and Chambers [11], each method implementation has a **predicate** expression which specifies when the method is applicable to a message send. The predicate can be an arbitrary boolean expression involving the arguments to the message send.

The extensible decision mechanisms of these two languages are quite similar: pointcuts and advice in AspectJ are analogous to predicates and methods in PD[1]. However, each mechanism has some advantages over the other. In AspectJ, a piece of advice can apply to more than one message, while a PD method must have a name that determines the message that it applies to. In AspectJ, pointcuts can access more information about a message send than just the data reachable from the receiver and arguments, including the control flow history and the location of the message send code. AspectJ has a form of method combination: the `before`, `after`, and `around` keywords determine whether a piece of advice runs before, after, or instead of the other code applicable to the join point, and in the body of an `around` method, a `proceed()` expression can be used to pass execution to the other code and intercept its return value; in PD, methods can't extend or combine with other methods, they can only override. On the other hand, AspectJ is tightly coupled to Java's single dispatch and has different syntax for advice and ordinary methods, both of which must be attached to classes, whereas PD is a natural generalization of OO multiple dispatch with just one kind of method, and methods can be declared as self-contained units apart from classes. In AspectJ, the rules for determining precedence between multiple pieces of advice that apply to the same join point are complicated and somewhat ad-hoc, involving aspect inheritance, an aspect dominance relation defined separately from inheritance, lexical ordering of the advice definitions in the code, and ordering of the source files in the argument list given to the compiler; method overriding in PD is based on logical implication of predicate expressions, which is a natural generalization of the method overriding rules in OOP languages (namely, subclass methods override superclass methods).

This paper presents a small prototype language called Fred that takes the best from both worlds. It has a dynamic dispatch mechanism that unifies those of AOP and OOP languages, and provides uniform support for incremental programming whether concerns are crosscutting or not. Decisions based on any information about the message send can be made extensible in Fred, including the message itself, the dynamic types and values of the message's arguments or any data reachable from them, the control flow history at the time of the message send, and the source location of the message send code. The decision criteria can be specified as arbitrary boolean expressions, but there is also syntactic sugar for a more declarative and concise style. Method combination is supported, and method overriding is determined by logical implication, although this can be customized if a different precedence relation is needed.

[1]The language described in [11] was not named, so I will use PD to refer both to the predicate dispatching mechanism and to that specific language.

Section 2 informally describes the syntax and semantics of Fred using a simple object-oriented example. Section 3 presents a longer example of a small data structure library that is incrementally extended with optimizations and new operations. Section 4 shows comparisons with other AOP languages. The paper concludes with a discussion of future research directions to be pursued.

2. FRED: A PROTOTYPE LANGUAGE OF EXTENSIBLE DECISIONS

In order to experiment with the idea of unifying aspect-oriented programming and predicate dispatching, I have developed a small prototype language called Fred, implemented as a set of procedures and macros in MzScheme [12]. I rely on a number of features of MzScheme to avoid having to reproduce them in the language design of Fred, such as first-class procedures and record datatypes (structures).

2.1 A Simple Example

There are three basic kinds of behavioral entities in Fred: **messages**, **branches**, and **decision points**. Messages and branches are defined with the special forms `define-msg` and `define-branch`; a branch has two closures over decision points, a predicate and a body. All behavior is performed in branch bodies, by sending messages to lists of arguments. Each message send causes a decision point entity to be created and processed. Decision points can be inspected with accessor functions such as `dp-msg`, `dp-args`, `dp-previous`, and `dp-source`. A simple example will serve to illustrate how these pieces can be put together:

```
(define-struct person
  (fname lname))
(define-struct (knight person)
  ())
(define-msg full-name)
(define-branch
  (lambda (dp)
    (and (eq? (dp-msg dp) full-name)
         (person? (car (dp-args dp)))))
  (lambda (dp)
    (let ((x (car (dp-args dp))))
      (string-append (person-fname x) " "
                     (person-lname x)))))
(define-branch
  (lambda (dp)
    (and (eq? (dp-msg dp) full-name)
         (knight? (car (dp-args dp)))))
  (lambda (dp)
    (string-append "Sir " (follow-next-branch))))
```

The first two definitions use MzScheme's structure definition syntax to create two structure types: `person`, with two fields `fname` and `lname`, and its subtype `knight`, with no additional fields. The `define-struct` form generates procedures for creating structure instances and accessing the fields, as well as a predicate procedure for testing whether an entity is an instance of the structure type; the name of the predicate is formed by appending a question mark to the type name. A type predicate also returns true for all instances of subtypes.

The third definition creates a new unique message entity and binds it to the name `full-name` in the current environment. Messages are implemented as procedures so that a message can be sent to a list of arguments simply by applying the message to the arguments.

The fourth definition creates a new branch to handle the `full-name` message being sent to a `person` value and adds it to the global table of branches. The first argument to `define-branch` is a condition predicate procedure that is applied to every decision point; if it returns true, then the second argument, the body procedure, is applied to the decision point, and its return value is returned as the value of the decision point. In this example, if the message of the decision point is equal to `full-name` and the first element of the argument list is an instance of the `person` type, then the `fname` and `lname` field values of the instance are concatenated and returned as the value of the decision point. In other words, sending the message `full-name` to a `person` instance evaluates to the string representation of the full name of the person.

The last definition creates a new branch to handle the special case of knights, by prepending "Sir" to the person's full name. When the `full-name` message is sent to a `knight` instance, both of the branches are applicable; when two or more branches apply to a given decision point, the branch whose predicate is most specific has higher precedence. Specificity is determined by logical implication: if a predicate p_1 implies another predicate p_2, i.e. p_2 is always true when p_1 is true, then p_1 is more specific than p_2. In this example, the second branch has precedence over the first, because `(knight? x)` implies `(person? x)`: all knights are also persons. The `follow-next-branch` procedure causes the next most precedent branch's body procedure to be invoked on the current decision point, and returns its return value. Thus, the second branch extends the behavior of the first branch in the case of the argument being a knight.

2.2 Ambiguous Message Sends

If Fred cannot determine a single most precedent branch for a given decision point, a "message ambiguous" error occurs. This can happen in one of three ways:

1. The predicates of two applicable branches both imply each other (i.e. they are logically equivalent).
2. Neither of the predicates of two applicable branches implies the other (i.e. they are logically independent).
3. Fred's implication determination algorithm fails on the predicates of two applicable branches.

In all of these cases, the user must disambiguate the two branches, either by explicitly declaring that one branch precedes the other or by creating a third applicable branch whose predicate is strictly more specific than the other two. (An example of the former is the `around` branch in Section 3.3, and an example of the latter is shown in the `empty-cord` extension in Section 3.2.)

The third case above can arise because 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.

2.3 Syntactic Sugar

Explicitly applying accessors to the decision point can be tedious, so there is some syntactic sugar available to make branch definition a bit more concise, similar to AspectJ's pointcut designator syntax:

```
(define-branch
  (&& (call full-name) (args person?))
  (with-args (x)
    (string-append (person-fname x) " "
                   (person-lname x)))))
(define-branch
  (&& (call full-name) (args knight?))
  (with-args (x)
    (string-append "Sir " (follow-next-branch))))
```

The `call`, `args`, `&&`, and `with-args` special forms all create procedures that take a decision point argument, so they can be used to define the two parts of a branch:

- The `call` form takes a list of messages and creates a predicate that tests a decision point's message for membership in the list.
- The `args` form takes a list of argument predicates and creates a predicate that applies each argument predicate to the corresponding element in a decision point's argument list.
- The `&&` form combines two decision point predicates into their conjunction. There are also `||` and `!` forms that create the disjunction and negation of decision point predicates, respectively.

- The `with-args` form takes a formal parameter list and a sequence of expressions and creates a body procedure that binds a decision point's argument values to the formals and evaluates the sequence in the resulting environment.

More syntactic sugar is available to make branch definition even more concise in many cases:

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

The `define-method` special form creates a branch whose predicate compares the message of a decision point to the given message, as well as providing names for the arguments to be bound to. It also defines the message if it is not already defined. Further syntactic sugar allows up to one test per parameter to be moved into the formals specifier, as long as the formal argument is named as the first operand of the test expression:

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

3. AN AOP EXAMPLE IN FRED

For a longer example, consider a library to implement **cords**, a data structure for strings that optimizes the concatenation operation by storing a tree of fragments rather than copying arrays of characters into a single array for every concatenation [2]. I will start with the basic structure and behavior and show how features and optimizations can be added to the library incrementally without modifying any code, using an aspect-oriented approach similar to the AspectJ implementation of cords by Huang [17].

3.1 Basic Structure and Behavior

First, we define three data types, `cord`, `flat-cord`, and `concat-cord`. `flat-cord` is just a wrapper around Scheme strings, while `concat-cord` has cords as left and right children; they both inherit from the abstract base class `cord`:

```
(define-struct cord ())
(define-struct (flat-cord cord)
  (string))
(define-struct (concat-cord cord)
  (left right))
```

Now we can start defining behavior over these types. First, the concatenation operation, which can handle both cords and strings for either argument, and produces a cord:

```
(define-method concat ((string? l) r)
  (concat (make-flat-cord l) r))
(define-method concat ((cord? l) (string? r))
  (concat l (make-flat-cord r)))
(define-method concat ((cord? l) (cord? r))
  (make-concat-cord l r))
```

Then we can define a length operator for the two kinds of cords:

```
(define-method len ((flat-cord? x))
  (string-length (flat-cord-string x)))
(define-method len ((concat-cord? x))
  (+ (len (concat-cord-left x))
     (len (concat-cord-right x))))
```

as well as an indexed reference operator:

```
(define-method ref ((flat-cord? x) (integer? i))
  (ref (flat-cord-string x) i))
(define-method ref ((concat-cord? x) (integer? i))
  (ref (concat-cord-left x) i))
(define-method ref ((concat-cord? x) (integer? i))
  & (>= i (len (concat-cord-left x)))
  (ref (concat-cord-right x)
       (- i (len (concat-cord-left x)))))
```

Note that the `ref` operator for `concat-cord` instances is split into two branches, one for each branch of the tree. The predicate of the second branch is more specific than that of the first branch, so it has precedence when they are both applicable.

3.2 Adding New Subtypes

We can add new subtypes to `cord` just as easily as in a traditional object-oriented language. For example, to optimize the substring operation, we can add a `substring-cord` type:

```
(define-struct (substring-cord cord)
  (base offset length))

(define-method substring ((string? b)
                          (integer? o)
                          (integer? l))
  (substring (make-flat-cord b) o l))
(define-method substring ((cord? b)
                          (integer? o)
                          (integer? l))
  (make-substring-cord b o l))
```

and add new branches for the existing operations when the first argument is a `substring-cord` instance:

```
(define-method len ((substring-cord? x))
  (substring-cord-length x))
(define-method ref ((substring-cord? x)
                    (integer? i))
  (ref (substring-cord-base x)
       (+ i (substring-cord-offset x))))
```

We can also add new subtypes that aren't implemented as structure types; for example, we can optimize the case of concatenating an empty cord to another cord, by defining an `empty-cord` predicate:

```
(define (empty-cord? x)
  (and (cord? x) (= (len x) 0)))

(define-method concat ((empty-cord? l) (cord? r))
  r)
(define-method concat ((cord? l) (empty-cord? r))
  & (not (empty-cord? l))
  l)
```

Note that the extra condition in the predicate of the second branch is required to ensure the two branches don't both apply when concatenating two empty cords; neither predicate is more specific than the other, so this would result in a "message ambiguous" error. Another way to avoid this would be to add a third branch to handle the overlap case explicitly:

```
(define-method concat ((empty-cord? l)
                       (empty-cord? r))
  l)
```

If a cord is built by successively concatenating very short strings, the tree will have many small nodes. We can improve performance if we coalesce these into larger nodes, by detecting this case and actually appending the strings rather than creating a new node each time:

```
(define *max-flat-len* 32)
(define (short-cords? l r)
  (and (flat-cord? l) (flat-cord? r)
       (< (+ (len l) (len r)) *max-flat-len*)))

(define-method concat (l r) & (short-cords? l r)
  (make-flat-cord (string-append
                   (flat-cord-string l)
                   (flat-cord-string r))))

(define-method concat ((concat-cord? c) r)
  & (short-cords? (concat-cord-right c) r)
  (make-concat-cord
   (concat-cord-left c)
   (concat (concat-cord-right c) r)))
```

The `short-cords?` predicate involves both of the arguments to the message send; this is something that type-based dispatch can't do, even with something like predicate classes [5] that allowed dynamic classification of each argument.

3.3 Adding New Crosscutting Code

Now suppose we want to optimize the cords library by keeping the tree structure balanced, so that the `ref` operator doesn't degenerate to linear search. This involves two things: keeping track of the depth of the tree, and rebalancing the tree after a node is added if the depth is too big. This code cuts across several of the previously defined operations and types, but we can still define it incrementally in Fred.

First, instead of modifying the data structures to add a `depth` field, we can create a new table and provide accessors that acts the same as field accessors would:

```
(define *depth-table* (make-hash-table 'weak))

(define (compound-cord? x)
  (or (concat-cord? x) (substring-cord? x)))

(define-method set-depth! ((compound-cord? x)
                           (integer? d))
  (hash-table-put *depth-table* x d))
(define-method depth ((compound-cord? x))
  (hash-table-get *depth-table* x))
(define-method depth ((flat-cord? x))
  0)
```

In order to add the `depth` field to multiple types at once, we make a new predicate that acts like a union type—again, without actually needing to implement a data structure for the type.

Now we need to extend the behavior of the compound cord constructors, `concat` and `substring`, to update the `depth` field and balance the tree if needed:

```
(define compound-cord-constructor?
  (|| (call concat) (call substring))

(define-around-branch compound-cord-constructor?
  (lambda (dp)
    (let ((x (follow-next-branch)))
      (update-depth! x)
      (ensure-balanced x))))
```

This branch is an **around** branch, a special kind of branch that has higher precedence than all non-around branches. Otherwise, because its condition is more general than the other branches that are applicable to `concat` and `substring` message sends, it would have the lowest precedence.

In order to actually update the depth of the compound cord, the decision needs to be split up into cases again:

```
(define-method update-depth! ((concat-cord? x))
  (set-depth! x
    (max (depth (concat-cord-left x))
         (depth (concat-cord-right x)))))
(define-method update-depth! ((substring-cord? x))
  (set-depth! x
    (+ 1 (depth (substring-cord-base x)))))
(define-method update-depth! ((cord? x))
  (void))
```

The third method is needed because `concat` can sometimes return non-compound cords, such as when one of the arguments is an empty cord. In this case the cord has no depth and nothing needs to be updated.

The code for balancing the tree similarly decomposes into cases:

```
(define-method ensure-balanced
      ((concat-cord? x)) & (unbalanced? x)
  (balance x))
(define-method ensure-balanced
      ((substring-cord? x)) & (unbalanced? x)
  (substring (balance (substring-cord-base x))
             (substring-cord-offset x)
             (substring-cord-length x)))
(define-method ensure-balanced ((cord? x))
  x)

(define (unbalanced? x)
  (< (len x) (fib (+ 2 (depth x))))))
(define-method balance ((concat-cord? x))
  ;; ...
  )
```

3.4 Vanishing Aspects

There is a problem with the definition of the `around` branch in the previous section. Notice that some of the `concat` and `substring` branches don't directly construct a new compound cord, namely when one of the arguments is a string instead of a cord; they convert the string into a cord, then re-send the message. In these cases, the `around` branch is followed twice, once for the original message send and then once for the recursive message send. The result is that the depth and balance check are computed redundantly.

One solution would be to change the condition of the `around` branch to only apply when none of the arguments is a string. However, suppose the implementation were changed so that compound cords could be composed of both cords and strings, and a `concat` message send would then always directly construct a `concat-cord` instance even if one of the arguments were a string. Then the `around` branch would not be followed at all in this case. This kind of situation is referred to as a "vanishing aspect" by Costanza [10], and is a dual of the more well-known problem of "jumping aspects" identified by Brichau, de Meuter, and de Volder [4].

A better solution is to change the condition to only apply to non-recursive message sends. The `dp-previous` accessor takes a decision point and retrieves the decision point that immediately precedes it in the stack of decision points being processed. The condition predicate of the `around` branch can use this to walk up the stack to make sure that there are no other compound constructor calls on it. This process can be abbreviated with the `cflow` special form, which takes a decision point predicate and creates a new decision point predicate that applies it to each previous decision point, returning true when it finds a match. So the around branch predicate can be replaced with:

```
  (&& compound-cord-constructor?
      (! (cflow compound-cord-constructor?)))
```

4. OTHER AOP LANGUAGES

While the design of Fred was mainly inspired by AspectJ, other AOP languages have similar mechanisms that support incremental programming with extensible decisions. I conjecture that the dispatch mechanism in Fred is basic enough to emulate most, if not all, of these other AOP languages. This section presents a survey of some of the more prominent AOP languages with brief discussions of how their mechanisms could be emulated with branches.

4.1 Composition Filters

ComposeJ [33] and Sina [21] allows message send decisions to be extended by attaching **composition filters** [1] to a class. All messages sent to instances of that class are intercepted by the filters, which may perform some action (such as delegating the message to some other code) based on predicate expressions being satisfied. Filters are essentially sets of branches whose predicates all refer to a particular class (or set of classes, with the superimposition mechanism), which can be parameterized.

4.2 Hyperslices and Hypermodules

Hyper/J [30] enables multi-dimensional separation and integration of concerns [31] by implementing the concerns as **hyperslice** that can be integrated based on specifications in **hypermodules**. A hypermodule has a set of instructions such as `merge`, `override`, and `bracket` that express different ways of combining the methods in multiple hyperslices into a set of output classes. A hyperslice is like a set of branches whose predicates can be parameterized by extra predicates in a hypermodule; for example, the `bracket` instruction can include a callsite specification that restricts the calling context, similar to a predicate that uses `dp-previous`. The hypermodule language provides finer-grained control over the precedence relation between branches, but everything must be specified explicitly, rather than having logical implication between predicates determine the default precedence relation.

4.3 Adaptive Programming

Adaptive programming [23] in DemeterJ [18] and DJ [26] allows the decision of what to do at each step of an object structure traversal to be extended by attaching an **adaptive visitor** to the traversal. Each visitor method specifies what behavior should be executed before, after, or instead of the traversal of objects of a particular class or the traversal through a particular member name. Wildcards can be used in the visitor method specification, for example to attach behavior to every object traversed, or to every member in the class graph with a given type. An adaptive visitor can be thought of as a set of branches whose predicates all refer to the same traversal, which is not determined until the visitor is attached to a particular traversal. Lieberherr, Patt-Shamir, and Orleans [24] discuss an extension that would allow a visitor method to only be executed when some condition on the current state and history of the traversal was true; this is similar to using `cflow` to distingush different paths in the call graph.

4.4 Aspectual Collaborations

Aspectual collaborations [22] provide a way to express a collaboration between several classes as a generic unit of behavior that can wrap new behavior around the methods of its participant classes. An **aspectual method** replaces another method (or set of methods) based on a pattern that matches the methods' static signatures. This is like a branch whose predicate refers to a set of messages and which is parameterized over the receiver class. Aspectual collaborations have the advantage that they can be separately typechecked and compiled; some of the techniques used in its implementation might be useful for optimizing branches that use a restricted subset of the predicate expression language.

4.5 Mixin Layers

Mixin layers [28] provide a way to express a generic collaboration as a set of **mixins** [3], which are essentially abstract subclasses, i.e. classes whose inheritance is parameterized. Each mixin contains a set of methods that can extend or override methods in other classes without specifying where those methods exist. A mixin layer can be thought of as a set of branches whose predicates refer to a set of messages and can be parameterized over the receiver argument classes.

4.6 Variation-Oriented Programming

Mezini's Rondo language [25] was designed to address context-dependent variations while supporting incremental programming. A Rondo program consists of a set of **adjustments**, which encapsulate sets of classes that extend other classes (in a generalized sense, without subtyping) when certain conditions hold. Adjustments are essentially sets of branches whose conditions share a common sub-condition.

5. CONCLUSION AND FUTURE WORK

This paper has shown how both AOP and PD languages provide better support for incremental programming than OOP by allowing extensible decisions that depend on information about a message send other than just the dynamic type of the receiver or arguments. A prototype language was presented with a dynamic dispatch mechanism that unifies those of AOP and OOP languages, and provides uniform support for incremental programming whether concerns are crosscutting or not. A running example demonstrated the features of the language, and comparisons with other AOP languages were shown. More research is needed, however, to better understand the language's dispatch mechanism, to extend it, and to build higher-level mechanisms on top of it to better support real-world programming.

In order to show that this model is basic enough to emulate other AOP systems, I plan to develop larger examples that compare directly to examples in those other systems, and perhaps develop translations from those systems into my model. For example, it should be possible to express all the examples from the AspectJ Programming Guide [32] in Fred, and either implement a translator from AspectJ to Fred or implement a set of macros that correspond to AspectJ syntax. This will probably involve extensions to the model, for example to emulate the `execution` primitive pointcut designator.

An obvious drawback to Fred's current implementation is that every decision point is processed dynamically, searching the global set of branches for the most specific applicable branch, which involves evaluating all the branch predicates. PD implementation techniques [6] can be used to improve the efficiency of this process by creating a dispatch tree at compile time that avoids repeating tests and has an optimal ordering of the tests. More structured support for expressing branch conditions, i.e. turning some of the syntactic sugar into primitive language constructs, will make programs more amenable to being statically analyzed and efficiently compiled—for instance, the `cflow` predicate can be much more efficiently implemented by putting marks on the stack at the point where its argument predicate is true, rather than actually walking up the stack at the point where the `cflow` predicate is evaluated.

A modularity mechanism is needed to organize branches into larger reusable components, just as methods are organized into classes and advice is organized into aspects. I have started to design a mechanism called **bundles** for this purpose, which are inspired by Flatt and Felleisen's units [13]. Units are reusable modules, parameterized by sets of import bindings and producing sets of export bindings. Units are linked together statically into compound units by connecting the imports and exports of other units together. Bundles generalize units by expanding the imports and exports to environments that include sets of branches in addition to variable bindings. Building parameterization directly into the module mechanism will lead to more flexible component composition than the abstract pointcut mechanism of AspectJ, which is too tightly coupled with Java's inheritance model.

The `around` branch mechanism allows the programmer to override the logical implication relation for determining branch precedence. A more complex customization mechanism might be needed in larger programs to better control the order of execution. One possibility is to include information about the module that the branch came from in the precedence relation, similar to how relationships between aspects in AspectJ determine the precedence of the advice in those aspects. Another possibility would be to allow a finer-grained mechanism for specifying the relation between two branches directly.

One feature that is common to both AspectJ and PD is the ability to bind variables in the pointcut or predicate that are then available to the body of the advice or method. This would be a useful addition to Fred as well; branches could then be parameterized over several different predicates that bind the same set of variables to different values extracted from the decision point. This would also remove code in the branch body that duplicates code in the predicate expression.

A common criticism of AOP is that it can be hard for someone reading the source code to determine exactly what behavior will occur for a particular message send; this was also discussed by Harrison and Ossher in the context of their **subdivided procedures** [16] language extension, which is a precursor to PD. In an OOP language, the same problem occurs, because the dynamic type of the receiver could be one of many different classes which are defined separately in the code, but the problem is exacerbated in Fred by the fact that predicates can be arbitrary expressions, so even if you know the dynamic types of the arguments you don't know where to look for the branch. AspectJ approaches this problem by providing smarter code browsers that can analyze the pointcuts and determine which aspects might be in effect at any particular line of code. A similar approach could be taken with Fred, perhaps using DrScheme's support for building development environments [7].

6. ACKNOWLEDGEMENTS

Many of the ideas in this paper were inspired by the discussion of incremental programming and context-dependent behavior variations in Mira Mezini's thesis [25]. The design of Fred was helped along by discussions with Craig Chambers and Greg Sullivan. The anonymous reviewers of the initially submitted version of this paper gave good advice about the order of presentation.

7. REFERENCES

[1] M. Aksit and B. Tekinerdogan. Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters. Technical report, TRESE project, University of Twente, Centre for Telematics and Information Technology, P.O. Box 217, 7500 AE, Enschede, The Netherlands, 1998. AOP'98 workshop position paper.

[2] H.-J. Boehm, R. Atkinson, and M. Plass. Ropes: an alternative to strings. *Software Practice & Experience*, 25(12):1315–1330, December 1995.

[3] G. Bracha and W. Cook. Mixin-based inheritance. In N. Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications / Proceedings of the European Conference on Object-Oriented Programming*, pages 303–311, Ottawa, Canada, 1990. ACM Press.

[4] J. Brichau, W. de Meuter, and K. de Volder. Jumping aspects. Workshop on Aspects and Dimensions of Concerns at ECOOP (position paper), Cannes, France, June 2000.

[5] C. Chambers. Predicate classes. In O. M. Nierstrasz, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 707, pages 268–296, Berlin, Heidelberg, New York, Tokyo, 1993. Springer-Verlag.

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

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

[8] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for java. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2000.

[9] W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. In N. Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 24, pages 433–444, New York, NY, 1989. ACM Press.

[10] P. Costanza. Vanishing aspects. Position paper, OOPSLA 2000 workshop on Advanced Separation of Concerns, October 2000.

[11] 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.

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

[13] 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.

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

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

[16] W. Harrison and H. Ossher. Subdivided procedures: A language extension supporting extensible programming. In *Proceedings: 1990 International Conference on Computer Languages*, pages 190–197. IEEE Computer Society Press, 1990.

[17] J. Huang. Experience using AspectJ to implement cord. Position paper, OOPSLA 2000 workshop on Advanced Separation of Concerns, October 2000.

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

[19] 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.

[20] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241, pages 220–242. Springer-Verlag, New York, NY, 1997.

[21] P. Koopmans. On the definition and implementation of the Sina/st language. MSc. thesis, Dept. of Computer Science, University of Twente, Enschede, the Netherlands, July 1995.

[22] 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.

[23] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996. ISBN 0-534-94602-X.

[24] K. J. Lieberherr, B. Patt-Shamir, and D. Orleans. Traversals of Object Structures: Specification and Efficient Implementation. Technical Report NU-CCS-02-02, College of Computer Science, Northeastern University, Boston, MA, February 2002.

[25] M. Mezini. *Variation-Oriented Programming Beyond Classes and Inheritance*. PhD thesis, University of Siegen, 1997.

[26] D. Orleans and K. Lieberherr. DJ: Dynamic adaptive programming in Java. In *Proceedings of Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns*, Kyoto, Japan, September 2001. Springer Verlag.

[27] A. Shalit. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language*. Addison-Wesley, Reading, Mass., 1997.

[28] Y. Smaragdakis and D. Batory. Implementing layered design with mixin layers. In E. Jul, editor, *Proceedings ECOOP'98*, pages 550–570, Brussels, Belgium, 1998.

[29] G. L. Steele. *Common Lisp the Language*. Digital Press, 2nd edition, 1990.

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

[31] P. L. Tarr, H. Ossher, W. H. Harrison, and S. M. S. Jr. N degrees of separation: Multi-dimensional separation of concerns. In *International Conference on Software Engineering*, pages 107–119, 1999.

[32] The AspectJ Team. *The AspectJ Programming Guide*. `http://aspectj.org/doc/dist/progguide/`.

[33] 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.