Incremental Programming with Extensible Decisions

Doug Orleans
College of Computer Science
Northeastern University
dougo@ccs.neu.edu

October 11, 2001

Abstract

This paper proposes a new model of programming, in which the behavior of a program can be defined as separate decision point branches. Allowing more precise expression of the condition determining when a branch should be chosen at a decision point leads to better support for incremental programming. This model can be viewed as a fundamental mechanism underlying both OOP and AOP, which can serve as lower-level building blocks that can be put together into the higher-level constructs present in many AOP systems.

1 Introduction

One of the chief benefits of object-oriented programming is the support for **incremental programming**, that is, the construction of new program components by specifying how they differ from existing components [CP89]: a subclass can extend or override the behavior of methods that exist in the superclass. Incremental programming allows for clean separation of concerns, because the differences of the new component can be specified separately from the existing component; this can lead to better understanding of programs, easier maintenance, and re-use.

When a subclass method extends or overrides a superclass method, essentially a new branch is added to a decision point in the program execution: if a message matching the methods' signature is sent to an instance of the subclass, then execute the subclass method instead of the superclass. However, the subclass-specific branch is defined separately from the superclass-specific branch, rather than tangled together into an explicit decision construct. As Mezini asserts in her thesis [Mez97][page 79], these extensible decisions are crucial:

> Avoiding the "case"-like style of procedural programming with respect to dispatching kind-specific behavior is an essential factor for the qualitative progress in reusability attributed to the object-oriented paradigm.

This idea can be extended to all of the decision points in a program, not just those that distinguish different kinds of objects. This paper proposes a new model of programming, in which the behavior of a program can be defined as separate decision point branches. Just as the behavioral atoms of procedural programming were split into the subatomic particles of methods in object-oriented programming, methods can be split into still smaller quantum units of behavior. This is achieved by allowing more precise expression of the condition determining when a branch should be chosen at a decision point. A method is a branch whose condition involves the run-time type of the receiver object of the message. Branch conditions should be able to involve other data as well: the state of the receiver object and the other message arguments; the message itself; the enclosing branch from which the message was sent; and global or thread-local properties of the system at the time of the message send.

Many behavioral design patterns [GHJV94] make these sorts of conditions expressible as extensible decisions by converting the decision into a dispatch based on receiver-type, because that's the only kind of dispatch that most object-oriented languages provide. The State pattern, for example, involves making a class for each different state that an object can be in, and sending a message to a state object to decide what behavior to execute based on the state object's class. However, it would be much more natural to express the state-based dispatch directly as a set of branches with the desired state conditions, without the conceptual and practical overhead of creating new classes and maintaining state objects.

Ideally the condition governing the applicability of a branch should be allowed to be any arbitrary computation involving any element of the program state at the decision point, although practical considerations such as tractability and efficiency limit this somewhat. Still, this goal can be approached by reifying decision points as program entities, and specifying branch conditions as logical combinations of predicate expressions over these decision point entities.

The next section describes a prototype language involving separately-specified branches with conditions that are predicate expressions over decision point entities. Following that is a discussion of related work, including connections to aspect-oriented programming. The paper concludes with a brief discussion of future research directions.

2 Fred: A Prototype Language of Extensible Decisions

In order to experiment with the idea of expressing behavior as extensible decisions, I have developed a small prototype language called Fred, implemented as a set of procedures and macros in MzScheme [Fla97]. 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, lexical scoping, and a structure mechanism.

There are three basic kinds of entities in Fred: **messages**, **branches**, and **decision points**. The programmer never creates decision points by hand, only messages and branches. Messages and branches are defined with the special forms `define-msg` and `define-branch`; decision points can be inspected in the code of a branch through the special variable `dp` with accessor functions such as `dp-msg` and `dp-args`. A simple example will serve to illustrate how these pieces can be put together:

```
(define-msg fact)
(define-branch (and (eq? (dp-msg dp) fact)
                    (= (car (dp-args dp)) 1))
  1)
(define-branch (eq? (dp-msg dp) fact)
  (let ((x (car (dp-args dp))))
    (* x (fact (- x 1)))))
```

The first expression creates a new unique message entity and binds it to the name `fact` 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 second expression creates a new branch to handle the base case of the factorial function and adds it to the global table of branches. The first subexpression of the `define-branch` special form is a condition predicate that is evaluated at every decision point; if it evaluates to true, then the remaining subexpressions in the form are evaluated. In this example, if the message of the decision point is equal to `fact` and the first element of the argument list is equal to 1, then the value 1 is returned as the value of the decision point. In other words, sending the message `fact` to the value 1 evaluates to 1.

The third expression creates a new branch to handle the alternative case; if the message `fact` is sent to any argument `x`, then `fact` is sent to `x-1` and the result is multiplied by `x`. Note that the predicates of both branches evaluate to true when the message `fact` is sent to 1; 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_1. In this example, the first branch is more specific than the second, because `(and X Y)` always implies `X`.

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:

```
(define-method fact (x) & (= x 1)
  1)
(define-method fact (x)
  (* x (fact (- x 1))))
```

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 fact ((= x 1))
  1)
```

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 [BAP95] [Hua00]. I will start with the basic structure and behavior and show how features can be added to the library incrementally without modifying any code.

First, we define three data types, `cord`, `flat-cord`, and `concat-cord`, using MzScheme's `define-struct` form. `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 struct:cord)
  (string))
(define-struct (concat-cord struct:cord)
  (left right))
```

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; this must be declared to Fred's implication system so that it can figure out when one predicate implies another:

```
(declare-implies 'flat-cord? 'cord?)
(declare-implies 'concat-cord? 'cord?)
```

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 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.

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 with new branches for the existing operations:

```
(define-struct (substring-cord struct:cord)
  (base offset length))
(declare-implies 'substring-cord? 'cord?)

(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)))
(declare-implies 'empty-cord? 'cord?)

(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 overlap (when concatenating two empty cords); neither branch 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)
```

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 re-balancing the tree after a concatenation if the depth is too big. We could modify the existing code to add these changes, but to achieve better separation of concerns, we should use the principle of incremental programming and implement the modification by expressing the new behavior as additions to the existing behavior. 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)))
(declare-implies 'concat-cord? 'compound-cord?)
(declare-implies 'substring-cord? 'compound-cord?)

(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 `concat` to update the depth field and balance the tree if needed:

```
(define-around (eq? (dp-msg dp) concat)
  (let ((c (invoke-next-branch)))
    (set-depth! c (max (depth (concat-cord-left c))
       (depth (concat-cord-right c))))
    (ensure-balanced c)))
(define-method ensure-balanced ((concat-cord? x))
  & (> (depth x) *max-depth*)
  (balance x))
(define-method ensure-balanced ((concat-cord? x))
  x)
```

The `invoke-next-branch` procedure invokes the branch that has the next highest precedence after the current branch. However, 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` message sends, it would have the lowest precedence.

3 Related Work

The branch model described in this paper is directly inspired by Ernst et al's predicate dispatching [EKC98]. In that system, each method has a predicate expression over the argument values, where the predicate expression language allows logical combinations of "is-a" tests or arbitrary test expressions in the base language. Method precedence is based on logical implication, where "is-a" atoms are compared based on the subtype relation. My branch conditions generalize method predicates by being expressions over decision points, which include the argument values but also the message value and calling context. Also, there is no method combination (a la `around` branches) in their system; there is no way to customize the method precedence relation. Predicate dispatching can be implemented efficiently [CC99], and the techniques for that can also be used to make decision point branches efficient.

More precise branch condition expression allows for finer granularity between behavioral units in a program, which in turn allows concerns to be more easily separated. In particular, concerns that would need to be tangled together with other concerns using traditional object-oriented design can be modularized well. This is also the goal of aspect-oriented programming [KLM$^+$97]. I view my model of branches and decision points as a fundamental mechanism underlying both OOP and AOP, which can serve as lower-level building blocks that can be put together into the higher-level constructs present in many AOP systems. The next few paragraphs discuss some particular AOP systems.

Aspects in AspectJ [KHH$^+$01] are units of crosscutting behavior; the behavior is specified as advice, which extend the behavior of classes which the aspect crosscuts. Message sends are reified as join point objects, which are roughly the same as decision points in Fred. Advice is defined in terms of pointcut designators, which specify sets of join points that the advice applies to; a pointcut designator is analagous to a branch condition expression, but the pointcut designator language is more declarative in flavor. It is also less general than predicate expressions: pointcuts cannot distinguish between different argument values, it can only make decisions based on run-time types (in addition to context information such as control flow).

Mezini's Rondo language [Mez97] 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.

Aksit's composition filters [AT98] extend message send decisions by attaching to a class filters that act on messages. All messages sent to instances of that class are first handled 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 conditions all refer to a particular class (or set of classes, with the superimposition mechanism), which can be parameterized.

4 Conclusion and Future Work

In this paper I have identified the fundamental mechanism of OOP that allows incremental programming, namely extensible message-send decisions. I have also shown how this mechanism can be made more flexible and that this leads to advanced separation of concerns mechanisms, thus prodiving a basic model of behavioral units that unifies OOP and AOP. More research is needed, however, to better understand this model, 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 [Tea] 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.

A modularity mechanism is needed to organize branches into larger components, just as methods are organized into classes and advice is organized into aspects. I have started to design something called **bundles** for this purpose, which are inspired by Flatt and Felleisen's units [FF98]. 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 tied up with Java's inheritance model.

In order for this model to improve the way programs are written, there needs to be a language that is as high-level as current OOP and AOP languages, as well as being comparably efficient. My prototype language is neither of these yet, but Scheme macros can help with the former, while predicate dispatch implementation techniques will improve the latter. A language that is not embedded and that has more structured support for branch conditions may better address both of these issues.

References

[AT98] Mehmet Aksit and Bedir 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.

[BAP95] Hans-Juergen Boehm, Russ Atkinson, and Michael Plass. Ropes: an alternative to strings. *Software Practice & Experience*, 25(12):1315–1330, December 1995.

[CC99] Craig Chambers and Weimin Chen. Efficient multiple and predicate dispatching. In *Proceedings of OOPSLA '99*, Denver, CO, November 1999.

[CP89] William Cook and Jens Palsberg. A denotational semantics of inheritance and its correctness. In Norman 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.

[EKC98] Michael D. Ernst, Craig Kaplan, and Craig 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.

[FF98] Matthew Flatt and Matthias 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.

[Fla97] Matthew Flatt. PLT MzScheme: Language manual. Technical report, Rice University, 1997.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, Massachusetts, 1994.

[Hua00] Jie Huang. Experience using AspectJ to implement cord. Position paper, OOPSLA 2000 workshop on Advanced Separation of Concerns, October 2000.

[KHH$^+$01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffery Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming*, 2001.

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241, pages 220–242. Springer-Verlag, New York, NY, 1997.

[Mez97] Mira Mezini. *Variation-Oriented Programming Beyond Classes and Inheritance*. PhD thesis, University of Siegen, 1997.

[Tea] The AspectJ Team. The AspectJ programming guide. Online manual. `http://aspectj.org/doc/dist/progguide/`.