

PROGRAMMING LANGUAGE SUPPORT FOR SEPARATION OF CONCERNS

A Thesis

Presented to the Faculty of the Graduate School
of the College of Computer Science
of Northeastern University
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by

Doug Orleans

July 2004

Abstract

The Socrates programming language unifies object-oriented and aspect-oriented language mechanisms into a few basic constructs plus layers of syntactic sugar. It supports advanced separation of concerns with open classes and predicate dispatching. Its predicate implication algorithm uses online partial evaluation with removal of redundant conditionals.

Chapter 1

Introduction

Edsger W. Dijkstra coined the term **separation of concerns** in 1976, and advocated it as a problem-solving principle: “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” [8, page 211]. In other words, break a problem down into easier, more manageable subproblems, and solve each of these smaller problems individually. This is a tried-and-true technique, and it’s only natural to apply it to all stages of software development, from analysis through design to implementation.

Analysis and design, however, are often done informally, with the output being documents that are only read by humans; the (natural) languages used are flexible enough to describe any desired separation of concerns. Implementation, on the other hand, must obey the rules of a formal programming language, so that the program can be interpreted by a computer. In order to make the implementation match the analysis and design as closely as possible, therefore, a programming language should be designed to provide maximal support for separation of concerns, with syntactic and semantic constructs that allow the different concerns to be expressed as separate program entities. This way, separation of concerns occurs not just in the problem-solving process but in the problem solution itself: the program code.

There are additional benefits to a program that exhibits separation of concerns. The most direct is that other humans reading the program can understand it in pieces, just like how the problem was

solved in pieces. Another is that if the program needs to be modified because of a change in the requirements of one concern, then the changes are likely to be localized to the code implementing that one concern. A corollary is that new concerns can be added to a program incrementally, and, with suitable abstraction mechanisms, code implementing a concern can be reused in other programs that share the same concern.

Object-oriented programming languages support separation of concerns with inheritance and polymorphism. Different concerns can be implemented as different classes such that one is a subclass or sibling class of another. An object-oriented filesystem implementation, for example, might put the code dealing with file access (permissions, timestamps, etc.) into an abstract `File` class, while the code dealing with the file contents would be separated into derived classes `PlainFile`, `Directory`, `SymLink`, etc.

Sometimes, however, concerns can't be separated into classes. In most single-dispatch object-oriented programming languages, a class serves as both a type and a module: all of the fields and methods associated with a class type must be defined as part of the class definition, even if they implement different concerns. Moreover, a single method might implement multiple concerns, because the concerns are associated with the same type and operation. Programmers often end up using design patterns [12] (e.g. State, Strategy, Visitor, or Command) to create artificial types in order to separate concerns into different classes.

In recent years the rise of aspect-oriented software development [10] (sometimes known as advanced separation of concerns) has brought new attention to designing programming languages to address these issues. Most of the major aspect-oriented approaches add new constructs, such as aspects [13], hyperslices [19], composition filters [1], or aspectual collaborations [17], to existing object-oriented programming languages, such as Java or Smalltalk. I have developed my own programming language, Socrates, that takes a different approach: it *replaces* object-oriented constructs with more general ones, and then provides syntactic sugar that can emulate both object-oriented and aspect-oriented constructs. This achieves a kind of separation of concerns in programming language design, between the core mechanisms and the programmer interface.

The two main features of Socrates supporting separation of concerns are **open classes** and **predicate dispatching**. With open classes, fields, methods, and inheritance relationships can be declared separately from their associated classes, and thus can be organized by concern rather than type. (An

open class is sometimes defined only as one whose methods can be declared separately [7]; my definition of open classes is referred to as **generalized open classes** by Clifton [6].) With predicate dispatching, the condition that determines whether a method is applicable can be arbitrary code [9]; thus, a method can be split into two or more methods associated with the same class, where each method implements a single concern.

Both of these features are present in AspectJ [13]: inter-type declarations allow fields, methods, and inheritance relationships to be declared in aspect definitions instead of inside the definition of their associated class, while the `if` pointcut designator allows arbitrary code to determine when a piece of advice is applicable. I don't claim that Socrates is any more expressive than AspectJ; my goal is a simpler language design that unites and generalizes several disparate constructs into a small number of easy-to-understand constructs, with more programmer flexibility due to the ease in which syntactic sugar can be added on top of the core language to emulate other methodologies.

My prototype implementation of Socrates is embedded in MzScheme [11] as a set of procedures and macros; thus it is a dynamically-typed, higher-order language with advanced scoping, modularization, and syntax extension capabilities. It would be possible to reimplement Socrates as a standalone language with (or without) these features, but MzScheme provides a convenient research platform.

This dissertation is organized as follows: part I serves as a Socrates language reference manual; part II presents some case studies of Socrates programming projects; part III analyzes the design and implementation of Socrates, compares it to related work, and explores some future directions in which this research could be extended.

Part I

The Socrates Programming
Language

Socrates is embedded in MzScheme [11]; it's implemented as a collection of libraries which contain a set of core procedures plus some layers of syntactic sugar to support different programming styles. The collection is called `socrates`, and individual libraries can be imported with a statement of the form `(require (lib library-file-name "socrates"))`. In particular, `socrates-core.ss` contains all the core procedures, while `socrates.ss` contains the entire Socrates language (procedures and syntax) and can be used as the initial import in a module declaration (in place of `mzscheme`):

```
(module module-identifier (lib "socrates.ss" "socrates")
  ; Socrates code goes here
)
```

Alternatively, the `socrates` collection can be specified on the command line with the `-M` flag to the `mzscheme` or `mred` executable, or Socrates can be selected in the language chooser of DrScheme [5].

Chapter 2 describes the core procedures provided by the `socrates-core.ss` library. Chapter 3 goes into further detail about the implication algorithm used to determine branch precedence in Socrates. Chapter 4 describes the different layers of syntactic sugar available in the full Socrates language.

Chapter 2

Core procedures

This chapter describes the `socrates-core.ss` library. It provides a set of core procedures (plus the `lambda/src` syntactic form) implementing the basic mechanisms of Socrates. A Socrates program could be written using only this library, but the syntactic forms provided by the `socrates.ss` library (described in chapter 4) provide a higher-level language interface. Some of the procedures provided by the `socrates-core.ss` library are replaced by syntax bindings in the `socrates.ss` library; in these cases, the underlying procedures are renamed with an asterisk (*) appended.

2.1 Messages

The fundamental operation in a Socrates program is sending a message. A **message** is a unique value representing an operation; it can be sent to a list of argument values to perform the operation on those values. The behavior of the operation is defined in branches (see section 2.3).

- `(make-msg)` returns a new message.
- `(msg? v)` returns `#t` if `v` is a message, `#f` otherwise.
- `(msg-send msg args)` sends `msg` to the `args` list. A message is also an applicable Scheme value, so it can be sent to a list of arguments directly with an application expression. In other words,

the following forms are all equivalent:

```
(msg arg ...)
```

```
(apply msg (list arg ...))
```

```
(msg-send msg (list arg ...))
```

In some object-oriented languages, a message consists of a selector and a list of arguments, and is sent to a single object, the receiver. In Socrates, a message is equivalent to a selector in these languages; its first argument can be considered the receiver, but it has no special status in determining what branch to execute. A message in Socrates is more like a generic function in CLOS [4] or other languages with multiple dispatch, except that while a generic function contains a set of methods, there is no explicit association between messages and branches.

2.2 Decision points

When a message is sent to a list of arguments, Socrates needs to decide what code to execute. This decision is based on the message, the arguments, and other dynamic context when the message was sent; this information is encapsulated into a **decision point**, which can then be processed and its contents dispatched to the appropriate code.

- (dp? *v*) returns #t if *v* is a decision point, #f otherwise.
- (dp-msg *dp*) returns the message being sent.
- (dp-args *dp*) returns the list of arguments.
- (dp-within *dp*) returns the branch currently being followed, or #f if the message was sent from the top level. Typically the body of this branch contains the message send expression that created *dp*, but it might be in some other Scheme procedure in the control flow of the branch body.
- (dp-previous *dp*) returns the decision point that caused the current branch to be followed, or #f if the message was sent from the top level.
- (dp-values *dp*) returns a list of the values returned by the branch that has most recently completed processing *dp*, or #f if no branches have yet been followed to completion.

- `(current-dp)` returns the decision point currently being processed.

Decision points cannot be created explicitly; they are only created when a message is sent.

2.3 Branches

The code to be executed when a message is sent is defined in separate **branches**. Each branch has a predicate which determines whether it should be followed given a decision point.

- `(make-branch pred-proc body-proc)` returns a new branch with predicate procedure *pred-proc* and body procedure *body-proc*. These should be Scheme procedures, both taking one argument: *pred-proc* takes a decision point, while *body-proc* takes whatever *pred-proc* returns. See section 2.3.1 for more details about how these are invoked.
- `(branch? v)` returns `#t` if *v* is a branch, `#f` otherwise.
- `(branch-pred branch)` returns the predicate of *branch*.
- `(branch-body branch)` returns the body of *branch*.
- `(current-branch)` returns the branch currently being followed.

A global dispatch table stores the set of active branches:

- `(add-branch branch)` adds *branch* to the dispatch table.
- `(remove-branch branch)` removes *branch* from the dispatch table.
- `(all-branches)` returns a list of all branches in the dispatch table. The order of the branches in the list may not be the same as the order in which they were added.

2.3.1 Predicates and bodies

When a message is sent, the decision point is passed as an argument to the predicate of every branch in the dispatch table. A branch predicate should return a true value if the branch is applicable,

or `#f` otherwise. If no branches in the dispatch table are applicable, the `exn:msg:not-understood` error is raised. Otherwise, among the branches whose predicates return a true value, the one with highest precedence is determined (see section 2.3.3). If there is no single applicable branch with higher precedence than the rest, the `exn:msg:ambiguous` error is raised; otherwise, the body of the most precedent branch is invoked, with its predicate’s return value as the body’s argument.

Because branch precedence (see section 2.3.3) is determined by inspecting the source expressions of predicate procedures, they should be created with the special form `lambda/src`, which records the source expression (and other information) but is otherwise identical to `lambda`. (The implementation of `lambda/src` is described in section 6.2.) The predicate procedure can contain any arbitrary Socrates code, but no guarantees are made about when and how often the procedure will be invoked. In addition, predicates should not contain side effects that affect the predicate’s value (e.g. mutating a local variable) because the implication algorithm ignores side effects.

2.3.2 Incremental behavior

A branch can provide incremental behavior by temporarily passing control in its body to the next most precedent applicable branch.

(`follow-next-branch filter-proc`) invokes the body of the next most precedent branch that is applicable to the current decision point and returns the value(s) returned by that invocation. The value that was returned by the next branch’s predicate (when it was invoked on the current decision point) is first passed to the procedure `filter-proc`, and its return value is passed to the next branch body. If there is no next most precedent branch, the `exn:msg:not-understood` error is raised; if there are multiple next most precedent branches, the `exn:msg:ambiguous` error is raised.

2.3.3 Branch precedence and around-branches

Branch precedence is determined by predicate implication: if the predicate p_1 of a branch b_1 logically implies the predicate p_2 of another branch b_2 —that is, p_2 can be proven to return true for all decision points for which p_1 returns true—then b_1 has precedence over b_2 . The algorithm for determining predicate implication is described in detail in section 3. Here’s a simple example demonstrating

branch precedence:

```
(define fact (make-msg))

(define fact-call?
  (lambda/src (dp)
    (and (eq? (dp-msg dp) fact)
         (car (dp-args dp)))))
(add-branch (make-branch fact-call? (lambda (n) (* n (fact (- n 1))))))

(define fact-base-case?
  (lambda/src (dp)
    (and (fact-call? dp)
         (zero? (car (dp-args dp)))))
  (add-branch (make-branch fact-base-case? (lambda (-) 1)))

(fact 5) ; => 120
```

The *fact-base-case?* predicate implies *fact-call?*, because (**and** *X Y*) implies *X* for any *X* and *Y*. Thus the second branch has precedence over the first when they are both applicable, namely when *fact* is sent to 0.

The precedence relation can be overridden by creating a special kind of branch called an **around-branch** that has precedence over all plain branches.

- (**make-around** *pred-proc body-proc*) returns a new around-branch with predicate procedure *pred-proc* and body procedure *body-proc*. These procedures should obey the same protocol as for plain branches described in section 2.3.
- (**around?** *v*) returns *#t* if *v* is an around-branch, *#f* otherwise. Around-branches are also branches, i.e. (**around?** *v*) implies (**branch?** *v*).

Precedence between around-branches is determined the same way as between plain branches, with one exception: if neither of the predicates of two around-branches implies the other (or both imply each other) then the more recently defined around-branch has precedence. In other words, around-branches never result in an ambiguous message error.

2.3.4 Run-time implication checking

Predicate implication and branch precedence can be computed statically, since they do not rely on the actual arguments of a message. Socrates also provides an interface for computing them dynamically at run time (e.g. in a reflective meta-predicate like those constructed by `subtype-of`—see section 4.4):

- `(pred-implies? pred-proc1 pred-proc2)` returns `#t` if `pred-proc1` can be proven to imply `pred-proc2`, or `#f` otherwise.
- `(branch-precedes? branch1 branch2)` returns `#t` if `branch1` has precedence over `branch2`, or `#f` otherwise.
- `(precedent-branches branches)` returns a list of branches in the `branches` list that have the highest precedence—in other words, the minima of the `branches` set according to the partial order defined by `branch-precedes?`.

Chapter 3

Predicate implication

When two plain branches or two around-branches are both applicable to a decision point, precedence between the two branches is determined by the implication relationship between their predicates: if one branch's predicate logically implies the other's, but not vice versa, then the first branch precedes the second. This chapter describes the algorithm used by Socrates to determine whether one predicate implies another. Since a decision point predicate can be any arbitrary Socrates expression, the algorithm can only approximate the implication relation, because it's undecidable in general whether one expression implies another in a Turing-complete language. The algorithm is conservative (assuming no side effects) in that it may report that a predicate does not imply another when in fact it does, but it will never report that a predicate does imply another when it doesn't. This is analogous to static type systems that may reject type-correct programs that it can't prove correct.

3.1 The implication algorithm



The predicate implication algorithm in Socrates is conceptually simple: a predicate $p1$ is considered to logically imply a predicate $p2$ when the following **implication procedure** is a tautology, i.e., always returns a true value:

```
(lambda/src args (or (not (apply p1 args)) (apply p2 args)))
```

In other words, when $p1$ and $p2$ are applied to the same arguments, either $p1$ returns false or $p2$ returns true.

A procedure is a tautology if its body expression is equivalent to a expression that is always true regardless of the procedure's arguments. This is determined by partially evaluating the procedure expression, with its formal arguments considered dynamic and its free variables considered static. The **lambda/src** form records an environment structure that maintains a mapping from the procedure's free variables to their current values; these values are used during partial evaluation to reduce fully-static expressions. In other words, Socrates uses an *online* partial evaluation strategy, similar to FUSE [18].

A partial evaluator for Scheme expressions is similar to a Scheme interpreter, except that it produces **partial values** rather than Scheme values. A partial value is either a Scheme value or a Scheme expression whose value can't be computed, either because it depends on dynamic arguments, or because the partial evaluator heuristically determines that it wouldn't terminate if the expression were evaluated.

3.1.1 Input syntax

The partial evaluator is a function from expressions to partial values. The input expression and all of its subexpressions should be in one of the following primitive syntactic forms:

- (**#%datum** . *datum*)
- (**quote** *datum*)
- (**quote-syntax** *datum*)
- *variable*
- (**let-values** (((*variable* ...) *expr*) ...) *expr*)
- (**letrec-values** (((*variable* ...) *expr*) ...) *expr*)
- (**if** *expr expr expr*)
- (**#%app** *expr* ...)

- (**case-lambda** (*formals expr*) ...)

(The **lambda/src** form records expressions in this syntax by expanding all macros and eliminating all side-effect-only expressions. See section 6.2 for more details.) The first three forms represent constant expressions; **#%datum** is inserted by MzScheme’s macro expander for literal constants (e.g. numbers and strings), while **quote** and **quote-syntax** are used by the programmer (usually abbreviated as **'** and **#'**) for non-literal constants (e.g. symbols and lists). The binding forms **let-values** and **letrec-values** are generalizations of **let** and **letrec** for expressions that can return multiple values. The application form **#%app** is inserted by MzScheme’s macro expander to represent non-macro applications. **case-lambda** is a generalization of **lambda** for procedures with multiple arities.

As an example of this primitive syntax, the implication procedure defined above will be macro-expanded into the following expression:

```
(case-lambda (args (let-values (((or-part) (%app not (%app apply p1 args))))
                    (if or-part or-part (%app apply p2 args)))))
```

3.1.2 Auxiliary data

During its recursive traversal of the expression, the partial evaluator also maintains some auxiliary data:

- a **partial environment** mapping variables to partial values, for evaluating applications and binding expressions;
- a list of closure cases currently being applied, to avoid infinite loops;
- lists of partial values assumed true and false, for reducing conditional expressions; and
- a flag that signals whether the expression is in a boolean context, in which case expressions which are known to be true will partially evaluate to **#t**.

At the top level (when partially evaluating an implication procedure), the partial environment is initialized from the procedure expression’s environment structure recorded by **lambda/src**, the

boolean context flag is true, and the other three lists are the empty list.

3.1.3 Partial evaluation rules

The partial evaluation function is recursively defined by the following rules:

1. Partially evaluating a constant expression (a `#%datum`, `quote`, or `quote-syntax` form) simply returns the constant value.
2. Partially evaluating a variable expression looks up the variable in the current partial environment to retrieve a partial value. This partial value is checked against the current assumptions, which may further reduce it (see section 3.1.6).
3. Partially evaluating a **let-values** binding expression causes the bound expressions to be partially evaluated and bound to the variables in an extended partial environment in which the body expression is partially evaluated. [TBD: explain multiple value binding]
4. Partially evaluating a **letrec-values** binding expression creates an extended partial environment in which the variables are bound to placeholder partial values, partially evaluates the bound expressions in this environment, replaces the placeholders with the resulting partial values, and then partially evaluates the body expression in this environment.
5. Partially evaluating a conditional expression (an `if` form) causes the test expression to be partially evaluated and checked against the current assumptions (see section 3.1.6), with the boolean context flag set to true. If the result is a true value, the consequent expression is partially evaluated; if the result is false, the alternative expression is partially evaluated; otherwise, the test partial value is added to the true assumptions while partially evaluating the consequent expression, then added to the false assumptions while partially evaluating the alternate expression, and finally the results are combined into a new conditional expression.
6. Partially evaluating an application expression (an `#%app` form) causes the operator and argument expressions to be partially evaluated; the operator is then **partially applied** to the list of argument values (see section 3.1.4) the result checked against the current assumptions (see section 3.1.6). [TBD: partial lists]

7. Partially evaluating an abstraction expression (a **case-lambda** form) returns `#t` if the boolean context flag is true; otherwise, it returns a **partial closure** value, which encapsulates the cases of the expression (formals forms and body expressions) and the current partial environment.

3.1.4 Partial application rules

Partially applying a partial value to a list of partial values is defined by the following rules:

1. If the partial value being applied is a partial closure, the applicable case is determined by matching the argument list's length to the arity of each case's formals form. If the applicable case is present in the list of cases currently being applied, an application expression is returned; otherwise, the case is added to that list while partially evaluating the body of the case in the case's partial environment extended with bindings for the formal variables to the corresponding argument partial values.
2. If the partial value being applied is a procedure whose source expression is available (because it was created by a **lambda/src** expression), that procedure expression is partially evaluated to a partial closure, which is then partially applied to the list of argument partial values according to rule 1.
3. If the partial value being applied is a procedure with an associated partial procedure available (see section 3.1.5), that partial procedure is applied to the list of argument partial values.
4. If the partial value being applied is a procedure and all the argument partial values are actual Scheme values (and there's at least one argument), then the procedure is applied to the list of argument values.
5. Otherwise, an application expression is returned.

3.1.5 Partial procedures

Some of Scheme's primitive procedures have associated **partial procedures**. A partial procedure is a procedure that takes some number of partial values and returns a partial value. For example, the `not` primitive has an associated partial procedure that checks its argument against the current

assumptions, returning `#f` if the argument is known to be true and `#t` if the argument is known to be false, and otherwise returning an application expression. A new partial procedure can be associated with a procedure using `(set-pproc! proc pproc)`.

3.1.6 Checking the current assumptions

Several of the rules in section 3.1.3 involve checking a partial value against the current assumptions. This may reduce the partial value into a constant true or false value or a simplified expression.

- If the boolean context flag is set and the partial value is a true value, an abstraction expression, or implied by one of the true assumptions (see section 3.1.7, then the partial value is replaced with the value `#t`.
- If the partial value is false, implies one of the false assumptions, or is disjoint from one of the true assumptions (see section 3.1.8, then the partial value is replaced with the value `#f`.

[TBD: distribute-if]

3.1.7 Implication between partial values

[TBD]

3.1.8 Disjointness of partial values

[TBD]



Chapter 4

Syntactic sugar

This chapter describes modules that provide a higher-level interface to the core procedures of Socrates. They are all included in the `socrates.ss` language module, or they can be required individually as needed.

4.1 Message sugar

The module `msg-sugar.ss` provides syntactic sugar for naming and defining messages.

`(make-msg)` is a syntactic form that returns a new message and infers its name from its context, similar to how MzScheme infers procedure names. (The underlying procedure is renamed to `make-msg*`.) For example, the following forms all set the created message's name to the symbol `'foo`:

- `(define foo (make-msg))`
- `(let ((foo (make-msg))) foo)`
- `((lambda (foo) foo) (make-msg))`

If a message's name cannot be inferred, it will be constructed using source location information. If source location is unavailable, the message's name will be 'anonymous.

(**msg-name** *msg*) returns the name of *msg*, or #f if *msg* was created by the **make-msg*** procedure directly.

(**set-msg-name!** *msg*) assigns a new name to *msg*.

(**define-msg** *name*) is equivalent to (**define** *name* (**make-msg**)).

(**ensure-msg** *name*) is equivalent to (**define-msg** *name*) if *name* is not already bound in any enclosing lexical scope (or the top level). Note that it cannot detect bindings in the *same* lexical scope—the following expression will cause an error, because it attempts to define *foo* twice:

```
(let ()
  (define-msg foo)
  (ensure-msg foo)
  foo)
```

4.2 Context sugar

The module `context-sugar.ss` provides syntactic sugar for passing a set of bindings from a branch predicate to a branch body.

(**make-context** (*name expr*) ...) returns a new **context instance** that binds each *name* to the corresponding expression *expr*.

(**lambda-context** (*name ...*) *body ...*¹) returns a new procedure that takes a single argument, a context instance, and binds each *name* to the value of the corresponding expression from the context instance when evaluating the *body* expressions. If one of the *names* was not bound in the context instance, an error is raised.

(**follow-next-branch** (*name expr*) ...) invokes the body procedure of the next most precedent branch with each *name* bound to the corresponding expression *expr*. (The underlying procedure is

renamed to `follow-next-branch*`.) These bindings override any bindings for the same names in the context instance returned by the next branch's predicate.

There is also a procedural interface to contexts:

- `(make-context* bindings-alist)` returns a new context instance from the association list *bindings-alist*, which should contain pairs of symbols and thunks (procedures with no arguments).
- `(context? v)` returns `#t` if *v* is a context instance, `#f` otherwise.
- `(context-bindings context)` returns the association list of bindings in the context instance *context*.
- `(context-value context sym)` returns the value of the expression associated with the symbol *sym* in the context instance *context* by invoking the corresponding thunk in the bindings association list. If *sym* has no binding, an error is raised.
- `(append-contexts context ...)` returns a new context instance containing all the bindings in the *contexts*. If the same name is bound in more than one *context*, the leftmost binding takes precedence in the new context instance. Argument values that are not context instances are ignored.

Thus, `(follow-next-branch (name expr) ...)` is equivalent to the following expression:

```
(follow-next-branch* (lambda (context)
  (append-contexts (make-context (name expr) ...) context)))
```

4.3 Predicate sugar

The module `predicate-sugar.ss` provides syntactic sugar for creating predicate procedures.

`(predicate formal-patterns [expr])` returns a new procedure (using `lambda/src`) whose body is the expression *expr* (or `#t` if not supplied). The *formal-patterns* form can take one of the following forms, similar to the formals form of a `lambda` expression:

- *(formal-pattern ...)*
- *(formal-pattern ...¹. variable)*
- *variable*

Each *formal-pattern* can take one of the following forms:

- *variable*
- *(pred-expr variable arg-expr ...)*
- **(bind-if** *(pred-expr variable arg-expr ...)* *(formal-pattern ...)*)

The second form acts as a specializer: the variable will be bound to the corresponding actual argument, and the form will be evaluated as an expression, i.e. the value of the *pred-expr* expression will be applied to the actual argument and the values of the *arg-expr* expressions; if it evaluates to false, then the whole procedure will return false. The third form is similar, but if the form evaluates to a context instance, variables in the *formal-patterns* will be bound to the values of the corresponding expressions in the context, and specializer forms in the *formal-patterns* will themselves be evaluated. The variables in the *formal-patterns* (including those nested in **bind-if** forms) are available to the body expression *expr*.

(return *(name expr) ...*) is equivalent to **(make-context** *(name expr) ...*).

(and/bind *bind-expr ...*) is an extension of Scheme's short-cutting **and** form to allow variables to be bound during the evaluation of the expressions and referenced in later expressions. Each *bind-expr* can take one of the following forms:

- *expr*
- **(begin** *expr ... bind-expr*)
- **(or** *bind-expr ...*)
- **(bind** *variable expr*)
- **(bind-if** *expr (formal-pattern ...)*)

The second and third forms are extensions of Scheme's **begin** and shortcutting **or** forms to allow *bind-expr* forms. The fourth form binds *variable* to the result of evaluating the expression *expr*; this binding is available to the remaining *bind-expr* forms in the enclosing **and/bind** expressions. The fifth form is similar to the **bind-if** formal pattern described above, but takes an arbitrary expression and makes the variables bound by the *formal-patterns* available to the remaining *bind-expr* forms in the enclosing **and/bind** expressions.

Note that the variables bound by the *bind-expr* forms are available to the remaining *bind-expr* forms in not just the same **and/bind** expression, but also *all enclosing and/bind* expressions. For example, the following expression is legal, even though *foo* is referenced outside of the **and/bind** expressions that bind it:

```
(and/bind (or (and/bind (bind foo (get-foo x)) foo)
              (and/bind (bind foo (get-foo y)) foo))
          (return (foo foo)))
```

Some useful predicates are also provided:

- (*? v ...*) always returns true; *?* can thus be used as a wildcard predicate.
- (*true? v*) returns *#t* if *v* is a true value, *#f* otherwise.
- (*false? v*) returns *#t* if *v* is false, *#f* otherwise.

As an example of the syntactic sugar defined in this section, the predicates from the example in section 2.3.3 could be rewritten as follows:

```
(define fact-call?
  (predicate (dp)
    (and (eq? (dp-msg dp) fact)
         (return (n (car (dp-args dp)))))))

(define fact-base-case?
  (predicate ((bind-if (fact-call? dp) ((zero? n))))))
```

4.4 Type constructors

A predicate can be considered to define a type, i.e. the set of (tuples of) values for which the predicate returns true; the module `type-constructors.ss` provides a set of higher-order procedures (plus the `&&` and `||` syntactic forms) that can be considered type constructors, because they create predicates.

`(&& type-expr ...)` is equivalent to `(predicate args (and (apply type-expr args) ...))`.

`(|| type-expr ...)` is equivalent to `(predicate args (or (apply type-expr args) ...))`.

`(! type)` returns `(predicate args (not (apply type args)))`.

`(opt type)` returns `(|| false? type)`.

`(list-of type)` returns a predicate that returns true if its argument is a list and `type` returns true for all its elements.

`(vector-of type)` returns a predicate that returns true if its argument is a vector and `type` returns true for all its elements.

`(pair-of type1 type2)` returns a predicate that returns true if its argument is a pair, `type1` returns true for its car, and `type2` returns true for its cdr.

`(alist-of type1 type2)` returns `(list-of (pair-of type1 type2))`.

`(eq v)` returns a predicate that returns true if its argument is `eq?` to `v`.

`(eqv v)` returns a predicate that returns true if its argument is `eqv?` to `v`.

`(equal v)` returns a predicate that returns true if its argument is `equal?` to `v`.

`(subtype-of type)` returns a predicate that returns true if its argument is a predicate `subtype` such that `(pred-implies? subtype type)` returns true.

4.5 Pointcut sugar

The module `pointcut-sugar.ss` provides syntactic sugar for decision point predicates similar to AspectJ's pointcut designator language [13].

`(call msg ...1)` is equivalent to `(predicate (dp) (or (eq? (dp-msg dp) msg) ...1))`.

`(args pred-expr ...)` returns a decision point predicate that applies the values of the *pred-expr* expressions to the arguments of the decision point (from left to right), and if they are all true, returns a context binding the name **args** to the list of arguments; if any return `#f`, or if there are too few arguments, the predicate returns `#f`. If the final *pred-expr* is the special identifier `..`, any number of additional arguments will be accepted; otherwise, the predicate returns `#f` if there are too many arguments.

`(bind-args formal-patterns [pred-expr])` returns a decision point predicate that binds the arguments of the decision point to the variables in *formal-patterns* (if there are the correct number of arguments), then evaluates the specializer forms in *formal-patterns* and the *pred-expr* expression (if supplied) with those bindings. (The syntax for *formal-patterns* is defined in section 4.3.) If they all return true, the predicate returns a context binding each variable in *formal-patterns* to the corresponding argument of the decision point; otherwise, the predicate returns `#f`.

[TBD: **with-vars**]

`(cflowbelow pred-proc)` returns a decision point predicate that returns true if *pred-proc* returns true for any of the decision point's previous decision points—i.e. any decision point reachable by one or more applications of `dp-previous` (see section 2.2).

`(cflow pred-proc)` returns `(|| pred-proc (cflowbelow pred-proc))`.

4.6 Branch sugar

The module `branch-sugar.ss` provides syntactic sugar for branches.

(**make-branch** *pred-expr body-expr*) and (**make-around** *pred-expr body-expr*) are syntactic forms that create a plain branch or an around-branch and store the *pred-expr* and *body-expr* expressions in place of the expression recorded by **lambda/src**. (The underlying procedures are renamed to **make-branch*** and **make-around***.)

(**make-before** *pred-expr body-expr*) is equivalent to

```
(make-around pred-expr (lambda/src (context)
                                   (body-expr context)
                                   (follow-next-branch)))
```

(**make-after** *pred-expr body-expr*) is equivalent to

```
(make-around pred-expr (lambda/src (context)
                                   (begin0
                                       (follow-next-branch)
                                       (body-expr context))))
```

(**define-branch** *pred-expr body-expr*) is equivalent to (**add-branch** (**make-branch** *pred-expr body-expr*)), except that it can appear in either an expression context or a definition context. **define-around**, **define-before**, and **define-after** are defined similarly, using **make-before**, **make-after**, **make-around**, respectively, in place of **make-branch**.

4.7 Method sugar

The module `method-sugar.ss` provides syntactic sugar for method-like branches similar to the syntax of CLOS [4] or the predicate dispatching language [9].

(**method-call** (*msg . formal-patterns*) [*pred-expr*]) is equivalent to

```
(&& (call msg) (bind-args formal-patterns [pred-expr]))
```

(**make-method** (*msg . formal-patterns*) [*& pred-expr*] *body-expr ...*¹) is equivalent to

```
(make-branch (method-call (msg . formal-patterns) [pred-expr]))
```

(**lambda-context** (*var* ...) *body-expr* ...¹)

where the *vars* are the variables in the *formal-patterns* form. The **&** is a special identifier to separate the predicate expression from the body expressions, similar to the **when** keyword in predicate dispatching.¹ **make-before-method**, **make-after-method**, and **make-around-method** are defined similarly.

[TBD: **with-vars**]

(**define-method** *name formal-patterns* [**&** *pred-expr*] *body-expr* ...¹) is equivalent to

```
(begin (ensure-msg name)
      (add-branch
       (make-method (name . formal-patterns) [& pred-expr] body-expr ...1)))
```

except that it can appear in either an expression context or a definition context. **define-before-method**, **define-after-method**, and **define-around-method** are defined similarly.

4.8 Field sugar

The module `field-sugar.ss` provides a datatype and syntactic sugar for fields and accessor branches.

A **field instance** is essentially just a table mapping values to values:

- (**make-field** [*default*]) returns a new field instance whose default value is *default* (or `#f` if not supplied).
- (**field?** *v*) returns `#t` if *v* is a field instance, `#f` otherwise.
- (**set-field-value!** *field key v*) associates *v* with *key* in the field instance *field*, replacing any previous association for *key*. Keys are compared with `eq?`. Keys are weakly held.
- (**field-value** *field key*) returns the value associated with *key* in the field instance *field*, or its default value if there is no such association.

¹MzScheme already has a **when** syntactic form, and I didn't want to overload it.

- `(field-default field)` returns the default value of the field instance *field*.

The Socrates naming convention for getters and setters is `get-field-name` and `set-field-name!`. `(getter field-name)` and `(setter field-name)` are syntactic forms that are equivalent to these names.

`(define-field-msgs field-name)` creates two messages and binds them to `(getter field-name)` and `(setter field-name)`. `(ensure-field-msgs field-name)` is similar but only creates the messages if their names are not already bound.

`(getter-call field-name)` and `(setter-call field-name)` are equivalent to `(call (getter field-name))` and `(call (setter field-name))`, respectively.

`(getter-method-call field-name type-expr)` is equivalent to

```
(method-call ((getter field-name) (type-expr key)))
```

`(setter-method-call field-name type-expr)` is equivalent to

```
(method-call ((getter field-name) (type-expr key) value))
```

`(make-getter-method field-name type-expr field)` is equivalent to

```
(make-branch (getter-method-call field-name type-expr)
              (lambda-context (key) (field-value field key)))
```

`(make-setter-method field-name type-expr field)` is equivalent to

```
(make-branch (setter-method-call field-name type-expr)
              (lambda-context (key value) (set-field-value! field key value)))
```

`(define-field field-name type-expr [default])` is equivalent to

```
(begin (ensure-field-msgs field-name)
       (let ((field (make-field [default])))
         (add-branch (make-getter-method field-name type-expr field)
                     (add-branch (make-setter-method field-name type-expr field))))))
```

except that it can appear in either an expression context or a definition context.

4.9 Objects and initializer sugar

The module `object-sugar.ss` provides a datatype for objects, plus some messages and branches defining a construction protocol. It also provides some syntactic sugar for defining initializer methods.

- `(make-object class)` returns a new **object instance** whose class is `class`. An object's class can be any value, but by convention it's a predicate procedure that returns true for the object.
- `(object? v)` returns `#t` if `v` is an object instance, `#f` otherwise.
- `(object-class object)` returns the class of the object instance `object`.

Field values can be associated with a value using the construction protocol defined by the `make`, `alloc`, and `init` messages:

- `(make pred-proc init-arg ...)` should return a value for which `pred-proc` is true. The default method for `make` calls `(alloc pred-proc init-arg ...)` to allocate a value `v`, then calls `(init pred-proc v init-arg ...)` to initialize `v`, then returns `v`.
- `(alloc pred-proc init-arg ...)` should return a value for which `pred-proc` is true. The default method for `alloc` calls `(make-object pred-proc)`.
- `(init pred-proc v init-arg ...)` should initialize (by side-effect) the value `v` with the `init-args` as initialization arguments. The default method for `init` does nothing, returning void.

`(init-method-call ((type-expr this) . formal-patterns) [pred-expr])` is equivalent to

`(method-call (init ((eq? t type-expr) this . formal-patterns)))`

`(make-init-method ((type-expr this) . formal-patterns) [& pred-expr] body-expr ...1)` is equivalent to

`(make-branch (init-method-call ((type-expr this) . formal-patterns) [pred-expr])
(lambda-context (var ...) body-expr ...1))`

where the *vars* are the variables in the *formal-patterns* form. **make-before-init-method**, **make-after-init-method**, and **make-around-init-method** are defined similarly.

(define-init-method ((*type-expr this*) . *formal-patterns*) [**&** *pred-expr*] *body-expr* ...¹) is equivalent to

(**add-branch** (**make-init-method** ((*type-expr this*) . *formal-patterns*) [**&** *pred-expr*] *body-expr* ...¹))

except that it can appear in either an expression context or a definition context. **define-before-init-method**, **define-after-init-method**, and **define-around-init-method** are defined similarly.

Field values can be retrieved from a value using the protocol defined by the `->list` message:

- `(->list v)` returns a list of the field values associated with the value *v*. The default method for `->list` when sent to an object calls `(->list (object-class v) v)`.
- `(->list class v)` returns a list of the field values associated with the value *v* for the class *class*. The default method when sent to `object?` and an object returns the empty list.

4.10 Predicate abstraction sugar

The module `predicate-abstraction-sugar.ss` provides syntactic sugar for defining **predicate abstractions** using messages and branches. A predicate abstraction is essentially a disjunction of predicates, but the predicates can be defined separately.

(make-predicate-msg) returns a new message and infers its name from its context, similar to **(make-msg)**. It also adds a default branch to the global dispatch table that returns false when this message is sent to any arguments. In other words, sending this message will never result in an `exn:msg:not-understood` error.

(define-predicate-msg name) is equivalent to **(define name (make-predicate-msg))**.

(ensure-predicate-msg name) is similar to **(ensure-msg name)** except that it also adds a default branch for the created message if *name* is not already bound.

(predicate-call (*msg . formal-patterns*) [*pred-expr*]) is similar to **(method-call** (*msg . formal-patterns*) [*pred-expr*]) except that instead of returning a context instance that binds the variables in *formal-patterns*, it returns the value of *pred-expr* if it's present (or **#t** otherwise).

(define-predicate *name type-expr*) is equivalent to

```
(begin (ensure-predicate-msg name)
       (define-branch (predicate-call (name . args) (apply type-expr args))
                     (lambda/src (context) context)))
```

(define-predicate (*name . formal-patterns*) [*pred-expr*]) is equivalent to

```
(begin (ensure-predicate-msg name)
       (define-branch (predicate-call (name . formal-patterns) [pred-expr])
                     (lambda/src (context) context)))
```

(define-predicates *formal-patterns* [& *common-pred-expr*] (*name pred-expr*) ...) is equivalent to

```
(begin
  (define-predicate (name . formal-patterns)
    (and/bind [common-pred-expr] pred-expr))
  ...)
```

In other words, it defines a set of predicate abstractions with the same formal patterns and a common predicate expression (whose bindings are available to each predicate abstraction body).

(classify *formal-patterns* [& *common-pred-expr*] (*name pred-expr*) ...) defines a set of predicate abstractions similar to **define-predicates**, but each predicate abstraction includes the negation of the previous predicate abstractions in this set. In other words, the defined predicates will be mutually exclusive.

(declare-implies *type supertype* ...) is equivalent to **(begin (define-predicate** (*supertype . args*) (apply *type args*) ..). In other words it extends each *supertype* predicate abstraction so that it also returns true when *type* returns true.

4.11 Class sugar

The module `class-sugar.ss` provides syntactic sugar for class-like predicate abstractions.

(make-class [*object->context*]) returns a new predicate abstraction that returns a true value if its argument is an object instance whose class is the predicate message. The true value will be the value returned by *object-¿context* when applied to the object, if provided, or `#t` otherwise.

(define-class *name* (*supertype* ...) (*field-name* ...)) defines a new class predicate, binding it to *name*, and a new field for each *field-name*. The class predicate returns a context instance binding each *field-name* to the current field value associated to the object. The class predicate is declared to imply each *supertype* predicate abstraction. A new initializer method for this class is defined that takes initialization arguments corresponding to the fields, plus some supertype initialization arguments; it forwards these to the first *supertype*'s initializer before calling each field's setter on the corresponding initialization argument.² A new method is added to `->list` for this class that appends the field values to the field lists of its supertypes.

4.12 Composition filters sugar

The module `filter-sugar.ss` provides syntactic sugar to emulate **composition filters** [1, 3]. A composition filter intercepts certain messages based on the **receiver** (the first argument) or the **sender** (the receiver of the previous message).

- **(dp-receiver** *dp*) returns the receiver of the decision point *dp*, or `#f` if the message was sent to no arguments.
- **(dp-sender** *dp*) returns the sender of the decision point *dp*, or `#f` if *dp* has no previous decision point or the previous message was sent to no arguments.

(define-input-filters *type filter* ...) defines a set of branches that act as a list of filters attached to *type*: when *type* is true for the receiver of any message, the decision point is processed by each

²A custom initializer method must be defined for a class with multiple supertypes requiring initialization arguments.

filter in sequence. (**define-output-filters** *type filter* ...) is similar, but the attached filters process decision points when *type* is true for the sender.

[TBD: *pred-call?*, *self-call?*, *class-call?*]

Each filter is specified as (*filter-handler filter-element* ...). The **filter elements** specify patterns for matching decision points; the **filter handler** specifies what action to take based on whether a decision point matches any of the patterns. Socrates provides a filter handler called **error-filter**, which raises an error if none of the patterns match, and otherwise passes the decision point on to the next filter. Other filter handlers can be defined; examples in the composition filters literature [1] include filters for delegation (**Dispatch**), synchronization (**Wait**), and reflection (**Meta**). A filter handler in Socrates is a procedure that takes a **message processor** (a predicate implementing the pattern match specified by the filter element list) and returns two values, an **accept predicate** and an **action procedure** that is invoked when the accept predicate is true. The action procedure takes one argument, the return value of the accept predicate. The error filter accepts a decision point when the message processor returns false; when a decision point is accepted, an exception is raised. Here is the simplified definition of the error filter handler:

```
(define (error-filter message-processor)
  (values
    (! message-processor)
    (lambda (-)
      (error "Message disallowed by filter."))))
```

Each filter element is specified as (*filter-operator condition msg* ...). The *filter-operator* is either the enable operator ($=>$) or the exclusion operator ($\sim>$). The *condition* is a decision point predicate, which usually inspects some data associated with the receiver. A filter element matches the decision point if its condition is true and the message of the decision point is in the provided list (or *not* in the provided list, with the exclusion operator).

Part II

Case studies of Socrates programs

In order to get a feel for “real world” programming with Socrates, I developed several small programming projects using Socrates. The idea was to go beyond toy programs to something that had interesting behavior besides just demonstrating some language feature. The following chapters describe these projects and evaluates them in terms of how Socrates helped with separation of concerns.

Chapter 5

Domino puzzle



The domino puzzle project was inspired by a logic puzzle that occasionally appears in Games magazine: use a set of dominos to cover a grid of numbers, where each domino must cover two adjacent numbers that match the domino. Using Socrates and PLT Scheme's MrEd graphical user interface toolkit [14], I developed a graphical application that randomly generates a (solvable) grid, displays it and a set of dominos, and allows the user to place dominos onto the grid by pointing and clicking.

The first separation of concerns is between the model and the view. The model concern is implemented by a set of classes and methods representing the board and the dominos, while the view concern is implemented by a set of methods that create MrEd widgets corresponding to objects in the model.

5.1 The model concern

The top-level class in the model is `polyomino-puzzle`. (I've generalized the problem from dominos to polyominos; the main procedure `make-domino-puzzle` just makes a `polyomino-puzzle` that happens to consist of dominos.) A `polyomino-puzzle` consists of a `polyomino-set` and a board. A `polyomino-set` is a set of polyominos. A polyomino is a matrix of tiles. A tile has a label, which can be any value, as well as a rotation, which is an integer between 0 and 3 inclusive, representing the number of 90°

```

(define-class polyomino-puzzle? () (polyomino-set board))
(define-class polyomino-set? () (polyominos))
(define-class polyomino? () (tile-matrix))
(define-class board? () (label-matrix tile-matrix))
(define-class tile? () (label rotation))

```

Figure 1: Class definitions for the domino-puzzle model.

rotations clockwise from “right-side up”. The labels in the default domino-set are simply integers, and rotation is ignored; however, an easier puzzle could be created where the labels were pictures, and you could only place a polyomino onto the board if the orientations of the pictures matched. A board consists of a matrix of labels (generated randomly) and a matrix of tiles (which starts out empty, i.e. #f in each cell). These five classes are summarized in figure 1.

The main operation for playing the puzzle is implemented by the *place!* method, which places a polyomino onto a board at a given position. (Position is an auxiliary class from the matrix module that encapsulates *x* and *y* coordinates.) The error-checking for this operation is separated into its own branch:

```

(define-method (place! (board? x) (position? pos) (polyomino? p))
  & (with-vars (msg) (bind-if (bad-placement? x pos p) (msg)))
  (error 'place! msg))

```

The *bad-placement?* predicate maps *bad-tile-placement?* over each tile in the polyomino. The latter predicate is specified with the help of a classifier (see figure 2).

5.2 The view concern

Each object in the model corresponds to a widget object in the view. These objects are connected by a two-way, one-to-one association, which is defined and initialized by the code in figure 3. There is also a type constructor *viewed* that distinguishes model objects that have views attached from ones that don't; for example, (*viewed board?*) is a predicate that returns true only for boards with views.

The top-level frame is created by sending *make-view* to a polyomino-puzzle object, which in turn

```

(define-predicate bad-tile-placement?
  (|| off-the-board? already-occupied? label-does-not-match?))

(classify ((board? x) (position? p) (tile? t))
  (off-the-board?
    (and (out-of-range? p (get-dimensions x))
      (return (msg (string-append "Off the board: " (->string p))))))
  (already-occupied?
    (and (occupied? x p)
      (return (msg (string-append "Already occupied: " (->string p))))))
  (label-does-not-match?
    (and (bind l1 (get-label x p)) (bind l2 (get-label t))
      (labels-do-not-match? l1 l2)
      (return (msg (format "Doesn't match: ~a ~a" l1 l2))))))

```

Figure 2: Branches of the `bad-tile-placement?` predicate.

```

(define-field model ?)
(define-field view ?)

(define-method (connect-view-model! v m)
  (set-model! v m)
  (set-view! m v))

(define-around-method (make-view model . rest)
  (let ((view (follow-next-branch)))
    (connect-view-model! view model)
    view))

(define (viewed type)
  (predicate (model) (and (get-view model) (type model))))

```

Figure 3: The view-model association.

sends *make-view* to each of its sub-objects. Thus each class in the model has a *make-view* method that constructs a widget.

The communication from the model to the view is done with advice, so that the model can be oblivious of the view. For example, whenever a polyomino is removed from a (viewed) polyomino-set, the polyomino-set view removes the corresponding child widget¹:

```
(define-before-method (rm! ((viewed polyomino-set?) x) (polyomino? p))
  (send (get-view x) delete-child (get-view p)))
```

Because I don't use drag-and-drop, placing polyominos onto the board is a two-step process: first click on a polyomino in the polyomino-set to select it, then click on a position on the board to place the polyomino. Thus the view concern requires remembering the current selection, but the selection concern can be defined independent of the view. I separated it out into the third main concern, sitting in between the model and the view.

5.3 The selection concern

Keeping track of the currently-selected tile in the puzzle is implemented with a field, which is set to #f when there is no selection:

```
(define-field selected-tile polyomino-puzzle?)
```

However, we need to keep track of more information to go along with the selected tile: we need to know which polyomino the tile is part of, what the tile's coordinates are relative to its polyomino, and whether that polyomino is in the domino-set or already on the board. The model doesn't have these backlinks, because it doesn't need them, but the selection concern needs to maintain them. It does so with advice on decision points that change the containment hierarchy (see figure 4). Using this support for selections, the selection concern defines a higher-level interface for manipulating the puzzle: *rotate-polyomino!*, *place-polyomino!*, and *remove-polyomino!* are messages that can be sent to a polyomino-puzzle object that act on the currently-selected polyomino. With the backlink

¹The **send** special form is from MrEd's object system, which is distinct from Socrates's; in particular, it's single-dispatch, so the receiver (the polyomino-set view) comes before the message (**delete-child**), followed by the message argument (the child to be deleted, i.e. the polyomino view).

```

(define-field polyomino tile?)
(define-field position tile?)

(define-after-method (set-tile! (polyomino? x) (position? p) (tile? t))
  (set-polyomino! t x)
  (set-position! t p))
(define-after (|| (method-call (set-tile-matrix! (polyomino? x) (matrix? m)))
  (method-call (rotate! (polyomino? x) (integer? i))))
  (lambda-context (x)
    (for-each-tile x (lambda (p t)
      (set-polyomino! t x)
      (set-position! t p))))))

(define-field parent polyomino?)
(define-after-method (set-polyominos! (polyomino-set? x)
  ((list-of polyomino?) l))
  (for-each (lambda (p) (set-parent! p x)) l))

(define-field position polyomino?)
(define-after-method (place! (board? x) (position? pos) (polyomino? p))
  (set-parent! p x)
  (set-position! p pos))

(define-field puzzle (|| polyomino-set? board?))
(define-after
  (|| (method-call (set-polyomino-set! (polyomino-puzzle? puzzle) (polyomino-set? x)))
  (method-call (set-board! (polyomino-puzzle? puzzle) (board? x))))
  (lambda-context (puzzle x)
    (set-puzzle! x puzzle)))

```

Figure 4: Fields and advice to maintain backlinks for the model containment hierarchy.

information, we can also define some predicates that partition tiles, polyominos, and polyomino-puzzles into subtypes (see figure 5), and use these predicates to define error-checking branches separately from the main operations (see figure 6).

```

(define-predicate (tile-on-board? (tile? x))
  (let ((polyomino (get-polyomino x)))
    (and (bind-if (polyomino-on-board? polyomino) (board))
         (return (polyomino polyomino) (board board))))))
(define-predicate (tile-off-board? (tile? x))
  (let ((polyomino (get-polyomino x)))
    (and (bind-if (polyomino-off-board? polyomino) (polyomino-set))
         (return (polyomino-set polyomino-set))))))

(define-predicate (polyomino-on-board? (polyomino? x))
  (let ((parent (get-parent x)))
    (and (board? parent)
         (return (board parent))))))
(define-predicate (polyomino-off-board? (polyomino? x))
  (let ((parent (get-parent x)))
    (and (polyomino-set? parent)
         (return (polyomino-set parent))))))

(classify ((bind-if (polyomino-puzzle? x) (polyomino-set board))
  (puzzle-with-selection?
   (and (bind tile (get-selected-tile x))
        tile
        (bind polyomino (get-polyomino tile))
        (return (polyomino-set polyomino-set) (board board)
                (selected-tile tile)
                (selected-polyomino polyomino)
                (selected-polyomino-parent (get-parent polyomino))))))
  (puzzle-with-no-selection?
   (return (polyomino-set polyomino-set) (board board))))))

(define-predicates ((bind-if (puzzle-with-selection? x)
  (polyomino-set board
   selected-tile
   selected-polyomino
   selected-polyomino-parent)))
  (puzzle-with-selection-on-board?
   (and (eq? selected-polyomino-parent board)
        (return (polyomino-set polyomino-set) (board board)
                (selected-tile selected-tile)
                (selected-polyomino selected-polyomino)
                (selected-polyomino-position
                 (get-position selected-polyomino))))))
  (puzzle-with-selection-off-board?
   (and (eq? selected-polyomino-parent polyomino-set)
        (return (polyomino-set polyomino-set) (board board)
                (selected-tile selected-tile)
                (selected-polyomino selected-polyomino))))))

```

Figure 5: Predicates that partition tiles, polyominos, and polyomino-puzzles into subtypes based on backlink information.

```
(define-branch (&& (call rotate-polyomino! place-polyomino! remove-polyomino!)
                  (args puzzle-with-no-selection? ..))
  (lambda (-)
    (error "Please choose a polyomino first.")))

(define-branch (&& (call rotate-polyomino! place-polyomino!)
                  (args puzzle-with-selection-on-board? ..))
  (lambda (-)
    (error "That polyomino has already been placed.")))

(define-branch (&& (call remove-polyomino!)
                  (args puzzle-with-selection-off-board? ..))
  (lambda (-)
    (error "That polyomino is not on the board.")))
```

Figure 6: Error checking for polyomino-puzzle operations.

5.4 Jabber library

[TBD]

Part III

Analysis

Chapter 6

Design notes and implementation details

6.1 Branch precedence

The branch precedence rule is the same as the method overriding rule used in the predicate dispatching system of Ernst, Kaplan, and Chambers [9]. They chose this rule because it subsumes and extends the usual object-oriented rule for overriding: subclass methods override superclass methods. For example:

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

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

The second *full-name* method precedes the first because *knight?* is a subclass of *person?*, i.e., the *knight?* predicate implies the *person?* predicate: if (*knight?* *x*) is true, then (*person?* *x*) is true, for

any x .

[TBD: around-branches]

6.2 Recording procedure source expression and environment

(**lambda/src** *formals expr ...*¹) is equivalent to (**lambda** *formals expr ...*¹), but it associates the source expression to the procedure, along with an environment structure for the free variables in the *expr* expressions. The expression is fully macro-expanded into primitive MzScheme syntax (except that top-level variables are not expanded to **#%top** forms), then regularized to a simpler syntax, removing all statements (expressions that are evaluated for side-effect only) along the way:

- (**case-lambda** (*formals stmt ... expr*) ...) is replaced with (**case-lambda** (*formals expr*) ...).
- (**lambda** *formals stmt ... expr*) is replaced with (**case-lambda** (*formals expr*)).
- (**if** *test then*) is replaced with (**if** *test then* (**#%app** void)).
- (**begin** *stmt ... expr*) is replaced with *expr*.
- (**begin0** *expr stmt ...*) is replaced with *expr*.
- (**let-values** (((*var ...*) *expr*) ...) *stmt ... expr1*) is replaced with (**let-values** (((*var ...*¹) *expr*) ...) *expr*). Note that binding forms with zero variables are removed.
- (**letrec-values** (((*var ...*) *expr*) ...) *stmt ... expr1*) is replaced with (**letrec-values** (((*var ...*¹) *expr*) ...) *expr*).
- (**letrec-syntaxes+values** *syntaxes* (((*var ...*) *expr*) ...) *stmt ... expr1*) is replaced with (**letrec-values** (((*var ...*¹) *expr*) ...) *expr*).
- (**set!** *var expr*) is replaced with (**#%app** void).
- (**with-continuation-mark** *key mark expr*) is replaced with *expr*.

The `proc-src.ss` module provides the following interface:

- `(proc/src? v)` returns `#t` if v is a procedure that was created with `lambda/src`, `#f` otherwise.
- `(proc-expr proc)` returns the expression (a syntax object) associated with procedure $proc$.
- `(set-proc-expr! proc expr)` associates the expression $expr$ with procedure $proc$. This only replaces the unexpanded expression; it's useful for recording syntactic sugar that expands into a `lambda/src` expression (which happens before `lambda/src` gets a chance to store it).
- `(proc-prim proc)` returns the regularized primitive-syntax expression (a syntax object) associated with procedure $proc$.
- `(proc-env-vars proc)` returns a list of the free variables (as symbols) of procedure $proc$.
- `(proc-env-value proc var)` returns the current value of the free variable var (a symbol) in the environment of procedure $proc$.

6.3 Predicate implication

- `(declare-prim-implies pred-proc1 pred-proc2)` asserts that $pred-proc1$ implies $pred-proc2$ (as well as all predicates that $pred-proc2$ implies). (This is a syntactic form so that it may appear in definition position, such as in the middle of a sequence of predicate definitions, but it operates on procedure values, not expressions.)

[TBD: partial evaluator notes]

[TBD: other pragmatics]

Chapter 7

Related work

7.1 Foundations

The research presented in this paper started with my observation of the similarities between the predicate dispatching language described by Ernst, Kaplan, and Chambers [9] and the dynamic join point model of AspectJ [13], particularly the notion of pointcuts as predicates. I also realized that predicate dispatching itself was a form of advanced separation of concerns, and it only needed to be extended a bit further to support crosscutting behavioral concerns [15]. I developed a prototype language called Fred [16] that implemented this basic extension, which evolved into Socrates as defined in part I. In this section I will summarize the differences between Socrates and predicate dispatching and between Socrates and AspectJ.

7.1.1 Predicate dispatching

Predicate dispatching “subsumes and extends object-oriented single and multiple dispatch, ML-style pattern matching, predicate classes, and classifiers, which can all be regarded as syntactic sugar for predicate dispatching” [9]. This is the philosophy behind the design of Socrates as well, which takes it a step further to subsume aspect-oriented mechanisms. In Socrates, predicates are functions

over decision points, rather than just functions over tuples of argument values. This generalization allows better separation of crosscutting concerns, because a predicate may accept multiple messages; in other words, a branch can cut across not only multiple types, but also multiple operations. In addition, a decision point includes the previous decision point and branch, which allows a predicate to distinguish between different control flows.

Ernst, Kaplan, and Chambers point out some limitations to their predicate implication algorithm:

We treat expressions from the underlying programming language as black boxes (but do identify those whose canonicalizations are structurally identical). Tests involving the run-time values of arbitrary host language expressions are undecidable. The algorithm presented here also does not address recursive predicates. While we have a set of heuristics that succeed in many common practical cases, we do not yet have a complete, sound, and efficient algorithm.

Socrates does not separate the predicate language from Scheme—predicates can contain an arbitrary mixture of Scheme and syntactic sugar defined by Socrates. Recursive predicates are allowed in Socrates, but the implication algorithm may fail when comparing them due to the termination heuristics of the implication algorithm (see section 3).

In Socrates, messages, predicates, and branches are first-class values which can be created and manipulated dynamically. Socrates also inherits Scheme’s lexical scoping. In predicate dispatching, all methods (and predicate abstractions) are declared statically by name at the top level. Also, predicate dispatching has no analogue to **follow-next-branch** in Socrates.

Named predicate abstractions are part of the core semantics of predicate dispatching; the main body of [9] only allows one definition per name, although an appendix mentions that multiple definitions could be combined with “or”. In Socrates, predicate abstractions are just syntactic sugar on top of messages and branches (see section 4.10). Furthermore, predicate dispatching assumes the host language has classes, but Socrates defines syntactic sugar for classes on top of predicate abstractions (see section 4.11).

One significant feature of predicate dispatching is not in Socrates: static typing. In predicate dispatching, methods and predicate abstractions can be declared to conform to type signatures,

allowing them to be checked at compile time for correctness, completeness, and uniqueness. The latter two guarantee that “message not understood” and “message ambiguous” errors, respectively, will never occur at run time. Socrates instead raises these errors at run time when the message is sent.

7.1.2 AspectJ

Decision points in Socrates were modeled after AspectJ’s join points. AspectJ has many different kinds of join points: method call, method execution, constructor call, constructor execution, object initialization, field reference, field set, advice execution, et.al. Most of these are not needed in Socrates, since object construction, initialization, and field reference are all done by sending messages. However, Socrates could share AspectJ’s distinction between method call and execution by differentiating between message send and branch body execution, but it doesn’t currently.

Predicate abstractions (section 4.10) in Socrates are similar to named pointcuts in AspectJ. However, named pointcuts cannot be extended; abstract pointcuts can be overridden, but this is actually just a parameterization mechanism. Context instances (section 4.2) in Socrates are similar to context exposure in AspectJ, except without the static type declarations; **follow-next-branch** can override context bindings like AspectJ’s **proceed**, although **proceed** is limited to the variables declared in the context exposure (which all must be present, even if they are not being overridden).

Branches in Socrates play the role of both methods and advice in AspectJ. Advice-like branches often need to be around-branches, though (see section 2.3.3), because their predicates cut across the predicates of other plain branches and are either less specific, identical to, or have no implication relationship with them. Advice is often additive, too, so multiple around-branches can exist without causing an ambiguous message exception; if there is no implication relationship between them, they are followed in an arbitrary order (most-recently-defined first). Advice precedence in AspectJ is not based on the contents of the pointcut at all, but is determined by relationships between aspects and lexical ordering in aspect definitions.

AspectJ has a form of open classes, allowing inter-type declarations of fields, methods, and inheritance relationships outside of class definitions. In Socrates, **define-field** (section 4.8), **define-method** (section 4.7), and **declare-implies** (section 4.10) can be used to define fields, methods,

and inheritance relationships outside of class definitions.

There is no special Socrates construct corresponding to aspects in AspectJ, but MzScheme's modularity structures (such as closures, modules, and units) can be used to encapsulate predicates, branches, fields, and inheritance relationships into something like an aspect.

7.2 Other AOP models

[TBD: compare with [21, 20]]

7.2.1 Composition filters

The composition filters model [1, 3] is an aspect-oriented technique where aspects are expressed as filters attached to a class (or a set of classes [2]) that intercept incoming and outgoing messages and process them according to some conditions. Socrates provides syntactic sugar for this style of programming; see section 4.12.

Figure 7 shows some example filter specifications from [1]; figure 8 shows the equivalent definitions in Socrates. Figure 9 shows a test program and its output.

```

USViewMail
inputfilters
USView: Error = {userView => {putOriginator, putReceiver, putContent,
                             getContent, send, reply},
                 systemView => {approve, putRoute, deliver},
                 true => {getOriginator, getReceiver, isApproved,
                          getRoute, isDelivered}};

ORViewMail
inputfilters
ORView: Error = {origView => {putOriginator, putReceiver, putContent,
                              getContent, send},
                 recView => {getContent, reply},*
                 true ~> {putOriginator, putReceiver, putContent, getContent,
                           send, reply}};

```

Figure 7: Example filter specifications from [1]. (The line marked with an asterisk was changed from `recView => reply`, which I believe was a mistake—a receiver should also be able to get the content of the mail.)

```

(define-input-filters user-system-view-mail?
 (error-filter (=> user-view?
                 set-originator! set-receiver! set-content!
                 get-content send reply)
 (=> system-view?
    approve set-route! deliver)
 (=> true?
    get-originator get-receiver get-approved?
    get-route get-delivered?))))

(define-input-filters originator-receiver-view-mail?
 (error-filter (=> originator-view?
                 set-originator! set-receiver! set-content!
                 get-content send)
 (=> receiver-view?
    get-content reply)
 (~> true?
    set-originator! set-receiver! set-content!
    get-content send reply))))

```

Figure 8: Definitions in Socrates corresponding to the examples in figure 7.

```

(define ana (make user? "Ana"))
(define bob (make user? "Bob"))
(define mailman (make user? "Mailman"))
(let ((mail (send ana bob "Hi there!")))
  (read bob mail)
  (let ((mail2 (reply bob mail "Hello back!")))
    (read ana mail2)
    ;; The nosy mailman tries to read the mail:
    (read mailman mail2)))

```

Ana sends mail to Bob.

Bob reads mail from Ana: "Hi there!"

Bob sends mail to Ana.

Ana reads mail from Bob: "Hello back!"

Message disallowed by filter: (get-content #<object:originator–receiver–view–mail?>)

Sender: Mailman

Figure 9: A program to test the filter definitions in figure 8.

Chapter 8

Conclusion and future work

[TBD]

Bibliography

- [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] L. Bergmans and M. Aksit. Composing multiple concerns using composition filters. Technical report, TRESE group, Department of Computer Science, University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands, 2001.
- [3] L. M. Bergmans. *Composing Concurrent Objects*. PhD thesis, TRESE, University of Twente, Enschede, The Netherlands, 1994.
- [4] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kicsales, and D. A. Moon. Common LISP object system specification X3J13 Document 88-002R. *SIGPLAN Not.*, 23(SI):1–143, 1988.
- [5] J. Clements, P. T. Graunke, S. Krishnamurthi, and M. Felleisen. Little languages and their programming environments. Monterey Workshop, 2001.
- [6] C. Clifton. Generalized open classes, multimethods, and modularity: Compilation, typechecking, and reasoning in MultiJava. Dissertation proposal, Jan. 2003.
- [7] 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.
- [8] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

- [9] 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.
- [10] R. E. Filman. A bibliography of aspect-oriented programming, version 1.22. Technical Report 03.01, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffett Field, California, June 2003.
- [11] M. Flatt. PLT MzScheme: Language manual. Technical Report TR97-280, Rice University, 1997.
- [12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.
- [13] 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.
- [14] MrEd web page: <http://www.plt-scheme.org/software/mred/>.
- [15] D. Orleans. Separating behavioral concerns with predicate dispatch, or, if statement considered harmful. In *Workshop on Advanced Separation of Concerns in Object-Oriented Systems (OOPSLA 2001)*, Oct. 2001.
- [16] 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.
- [17] 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.
- [18] E. S. Ruf. *Topics in online partial evaluation*. PhD thesis, Stanford University, 1993.
- [19] P. Tarr and H. Ossher. *Hyper/J User and Installation Manual*. IBM T. J. Watson Research Center, Yorktown Heights, NY, USA, 2000.
- [20] D. B. Tucker and S. Krishnamurthi. Pointcuts and advice in higher-order languages. In M. Akşit, editor, *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003)*, pages 158–167. ACM Press, Mar. 2003.

- [21] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. In *FOAL 2002: Foundations of Aspect-Oriented Languages (AOSD-2002)*, pages 1–8, Mar. 2002.