

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

August 2005

Abstract

The Socrates programming language unifies and generalizes object-oriented and aspect-oriented language mechanisms that support separation of concerns. In particular, methods and advice can be viewed as special cases of the same general mechanism: units of behavior that specify what to do (the body) and when to do it (the predicate). A program in the Socrates core language consists of messages, representing abstract operations, and branches, representing concrete units of behavior that implement parts of one or more operations. Both the body and the predicate of a branch can be unrestricted procedures, allowing the program code to be flexibly organized by concern, rather than only by type or operation. Precedence between branches whose predicates overlap is determined by logical implication analysis (a generalization of inheritance-based object-oriented precedence rules) plus mechanisms for overriding this default precedence relation (to support crosscutting behavior). Socrates also provides layers of abstraction on top of these primitive constructs to emulate features of other languages, including multimethods, predicate dispatching, generalized open classes, pointcuts and advice, structural pattern matching, classifiers, and composition filters.

Socrates is implemented as a language embedded in PLT MzScheme, so that Socrates code and Scheme code can be freely intermixed; in addition, messages are Scheme procedures and branches are first-class values that can be dynamically added to and removed from the global dispatch table. This dissertation presents the specification of this embedding, as well as a case study of a non-trivial GUI application implemented in Socrates, a formulation of the Law of Demeter for Socrates including a Socrates program to dynamically check for violations, and a partial-evaluator-based algorithm to analyze predicate implication. The Socrates implementation is a free software project on SourceForge (<http://socrates-lang.sf.net/>).

Contents

Abstract	3
1 Introduction	11
1.1 Separation of concerns	11
1.2 The Socrates programming model	15
1.3 Outline and research contributions	19
I The Socrates programming language	21
2 Core procedures	25
2.1 Messages	25
2.2 Decision points	26
2.3 Branches	27
2.3.1 Predicates and bodies	28
2.3.2 Incremental behavior	29

2.3.3	Branch precedence and around-branches	29
2.3.4	Computing implication and precedence	31
2.4	Fields and objects	31
2.5	Contexts	34
3	Predicate implication analysis	37
3.1	Canonicalization	38
3.2	Implication of canonicalized expressions	40
3.3	Primitive implication and disjointness	41
3.4	Primitive tautologies	42
3.5	Syntactic equivalence	43
4	Syntactic sugar	45
4.1	Message sugar	45
4.2	Context sugar	47
4.3	Predicate sugar	48
4.4	Type constructors	51
4.5	Pointcut sugar	52
4.6	Branch sugar	53
4.7	Method sugar	54
4.8	Field sugar	55

<i>CONTENTS</i>	7
4.9 Initializer sugar	56
4.10 Predicate abstraction sugar	57
4.11 Class sugar	58
4.12 Composition filters sugar	59
II Case studies of Socrates programs	63
5 Domino puzzle	67
5.1 The model concern	69
5.2 The view concern	71
5.3 The selection concern	72
5.4 Analysis	72
6 Law of Demeter checker	79
6.1 The Law of Demeter for Socrates	79
6.2 Checking the Law of Demeter in Socrates	81
6.3 Checking the domino puzzle code	86
6.4 Comparison with AspectJ implementation	87
III The Socrates implementation	89
7 Procedure and message reflection	93

7.1	Procedure reflection	93
7.2	Message reflection	96
8	Predicate implication analysis algorithm	99
8.1	Input syntax	100
8.2	Partial values	100
8.3	Auxiliary data	102
8.4	Partial evaluation	102
8.5	Distribution across conditionals	104
8.6	Partial application	105
8.7	Checking the current assumptions	106
8.8	Partial procedures	107
8.9	Memoization	109
IV	Analysis	111
9	Related work	113
9.1	Foundations	113
9.1.1	Predicate dispatching	113
9.1.2	AspectJ	119
9.2	Other AOP models	122

9.2.1	Hyperspaces	122
9.2.2	Composition filters	124
9.2.3	Classpects	124
9.3	Aspect calculi	124
9.3.1	μ ABC	126
9.3.2	MinAML	127
9.3.3	PAC	127
9.3.4	Extended CEKS machine	128
9.4	Partial evaluation	129
10	Conclusion and future work	131
10.1	Standalone language	131
10.2	Formal semantics	132
10.3	Branch precedence	133
10.4	Efficient implementation	133
10.5	Tool support	134

Chapter 1

Introduction

This dissertation presents the design and specification of the Socrates programming language and an implementation embedded in PLT MzScheme [23]. Socrates unifies and generalizes object-oriented [57] and aspect-oriented [36, 22] language mechanisms that support separation of concerns [16]. The design of Socrates is based on predicate dispatching [20], which itself unifies and generalizes multimethods [44], ML-style structural pattern matching [43], predicate classes [10], and classifiers [32]. Socrates further generalizes predicate dispatching to include aspect-oriented pointcuts and advice [35], composition filters [67], and generalized open classes [13]. The purpose that unites these many programming language mechanisms is the concept of separation of concerns.

1.1 Separation of concerns

Edsger W. Dijkstra advocated the following problem-solving principle in 1974: “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 [...] It is what I sometimes have called the **separation of concerns**” [16, 17, 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 is 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, the same way that 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. For example, consider the class definitions in figure 1 implementing a simple file system in Java [30]. The code dealing with inode allocation is in the abstract `File` class, while the code dealing with file contents is separated into the derived classes `RegularFile` and `Directory`.

Sometimes, however, concerns cannot 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 resort to using design patterns [28] (e.g. State, Strategy, Visitor, or Command) to create artificial types in order to separate concerns into different classes.

```

abstract class File {
    int inode; static int nextInode = 0;
    File() { inode = nextInode++; }
    abstract int getSize();
}

class RegularFile extends File {
    byte[] contents;
    byte[] getBytes(int offset, int length) {
        byte[] bytes = new byte[length];
        System.arraycopy(contents, offset, bytes, 0, length);
        return bytes;
    }
    int getSize() { return contents.length; }
}

class Directory extends File {
    java.util.Map entries;
    File getEntry(String name) { return (File) entries.get(name); }
    int getSize() {
        int size = 0; java.util.Iterator i = entries.values().iterator();
        while (i.hasNext()) size += ((File) i.next()).getSize();
        return size;
    }
}

```

Figure 1: An example of object-oriented separation of concerns.

In recent years the rise of aspect-oriented software development [36, 22, 4] (sometimes known as advanced separation of concerns) has brought new attention to designing programming languages to address these issues by supporting the separation of concerns that are not necessarily related to a particular type. For example, consider adding timestamping and permission checking features to the file system example from figure 1; using plain Java, code to implement these concerns would have to be included in the three class definitions. Figure 2 shows how these two concerns can be separated out into two aspects in AspectJ [35].

Most of the major aspect-oriented approaches add new constructs, such as aspects [35], hyperspaces [58], composition filters [3], or aspectual collaborations [50], to existing object-oriented programming languages, such as Java [30] or Smalltalk [29]. Socrates 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

```

aspect TimeStamping {
    long File.lastUsed = System.currentTimeMillis();
    before(File file):
        (call(* getBytes(..)) || call(* getEntry(..)) || call(* getSize(..)))
        && target(file)
    {
        file.lastUsed = System.currentTimeMillis();
    }
}

aspect PermissionChecking {
    boolean File.isReadable = true;
    before(File file):
        (call(* getBytes(..)) || call(* getEntry(..)))
        && target(file)
        && if (!file.isReadable)
    {
        throw new RuntimeException("Access denied.");
    }
}

```

Figure 2: An example of aspect-oriented separation of concerns.

separation of concerns in programming language design, between the core mechanisms and the programmer interface. This approach stems from the observation that methods and advice can be viewed as special cases of the same general mechanism: units of behavior that specify what to do (the body) and when to do it (the predicate).

The two main features of Socrates supporting separation of concerns are **predicate dispatching** and **open classes**. With predicate dispatching, the condition that determines whether a method is applicable can be arbitrary code [20]; thus, a method can be split into two or more methods associated with the same class, where each method implements a single concern. 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 [14]; this wider definition of open classes is referred to as **generalized open classes** by Clifton [13].)

Both of these features are present in AspectJ [35]: 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

```

(define-class file? () (inode))
(define next-inode 0)
(define-init-method ((file? x))
  (init file? x next-inode)
  (set! next-inode (+ next-inode 1)))
(define-msg get-size)

(define-class regular-file? (file?) (contents))
(define-method (get-size (regular-file? x))
  (bytes-length (get-contents x)))
(define-method (get-bytes (regular-file? x)
  (integer? offset)
  (integer? length))
  (subbytes (get-contents x) offset (+ offset length)))

(define-class directory? (file?) (entries))
(define-method (get-entry (directory? x) (string? name))
  (hash-table-get (get-entries x) name (lambda () #f)))
(define-method (get-size (directory? x))
  (apply + (hash-table-map (get-entries x)
    (lambda (name file)
      (get-size file)))))

```

Figure 3: The classes from figure 1 implemented in Socrates.

a piece of advice is applicable. Socrates is not necessarily any more expressive than AspectJ; its goal is to be 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. As an example of the expressiveness of this syntactic sugar, figures 3 and 4 show a Socrates implementation of the file system example from figures 1 and 2.

1.2 The Socrates programming model

A program in the Socrates core language consists of **messages**, representing abstract operations, **branches**, representing concrete units of behavior that implement parts of one or more operations, and **fields**, representing associations between data values. During the execution of a Socrates program, **decision points**, **objects**, and **contexts** are created and manipulated. These six entities are all first-class values.

```

(define-field last-used file?)
(define-after-init-method ((file? x) . _)
  (set-last-used! x (current-inexact-milliseconds)))
(define-before (&& (call get-size get-bytes get-entry)
  (bind-args ((file? x) . _)))
  (lambda-context (x)
    (set-last-used! x (current-inexact-milliseconds))))
(define-field readable? file? #t)
(define-before (&& (call get-bytes get-entry)
  (bind-args ((file? x) . _)
    (not (get-readable? x))))
  (lambda-context ()
    (error "Access denied.")))

```

Figure 4: The aspects from figure 2 implemented in Socrates.

The basic operation in Socrates is to send a **message** to a tuple (list) of arguments. A message represents an abstract operation, roughly analogous to a generic function in CLOS [7], and sending a message is analogous to applying a generic function. Whenever a message is sent, a decision is made about what to do based on information about the message, the arguments, and the current state of the system. This information is encapsulated into a structure called a **decision point**. Decision points are roughly analogous to AspectJ's join points, but the name emphasizes that there is a decision to be made about what code to execute.

The different choices of code to execute are partitioned into structures called **branches**. Similar to the branches of a conditional if or **switch** statement, these represent the alternatives that can be followed depending on some condition. Each branch contains its condition, in the form of a predicate procedure; the predicate takes a decision point as its argument, and returns true or false depending on whether or not the branch should be followed for that decision point. Unlike the branches of conditional statements, however, branches in Socrates are independent entities, and can be dynamically created and added to (or removed from) the global **dispatch table** that is consulted for each decision. The code to add branches can be organized into modules according to the concerns that the branches address, rather than having to be organized by type or operation.

Another key difference from the branches of a **switch** statement is that when more than one branch's condition predicate applies to a decision point, the precedence is determined not by

lexical ordering of the branches but according to relationships between the condition predicates. More specifically, if one branch's predicate X logically implies another branch's predicate Y —that is, whenever X is true, then Y must also be true—the first branch takes precedence. (If neither predicate implies the other, or both predicates are equivalent, an ambiguous message error is raised.) This is a generalization of the usual object-oriented method-overriding relation—subclass methods override superclass methods—because a subclass test logically implies a superclass test: an instance of a subclass is always an instance of its superclasses. Because branch predicates can be arbitrary procedures, the implication relation is undecidable in the general case, but the relation can be approximated using an analysis of the syntactic structure of the predicate expressions as well as knowledge about logical relationships between primitive procedures.

While predicate implication is a natural and useful way to determine branch precedence, it can sometimes get in the way of the separation of “crosscutting” concerns [36]. For example, a branch that handles an error condition for an operation may have a predicate that does not imply any single other branch that implements some part of the operation, but it must be given higher precedence than all of those branches in order to handle the error and not cause an ambiguous message error. One way to accomplish this would be to split it up into multiple branches such that each one had precedence over one other branch; another would be to rewrite the operation branch predicates to exclude the error condition, so that only one branch could apply to any decision point for that operation. Both of these approaches weaken the separation between the error and operation concerns, however, by mixing parts of the concerns in some of the branch predicates. In Socrates, the error branch can be made an **around-branch**, a special kind of branch that always has precedence over all plain branches. Precedence between two around-branches is still determined by predicate implication analysis, but if neither predicate implies the other or both are equivalent, the around-branch most recently added to the dispatch table has precedence rather than an ambiguous message error being raised. (This could be generalized into more than just two kinds of branches, or there could be other finer-grained mechanisms for controlling branch precedence, but in practice this simple mechanism is enough to support a large category of concern separation that cannot be done with just the usual object-oriented method-overriding relation.)

In a `switch` statement, one branch can “fall through” to the next branch; this is a crude mechanism to allow code to be shared among branches, with the upper branch supplying an incremental extension to the shared behavior in the lower branch. Socrates provides a more flexible mechanism: a branch can call the special procedure `follow-next-branch`, which will pass control to the next-most-precedent branch and then return to the current branch when it returns. This allows a branch to provide incremental behavior before or after the next branch, or even to follow the next branch multiple times (or not at all). The values provided to the next branch can be modified or replaced, as can its return value.

The primitive data structure entities in Socrates are objects and fields. An **object** is simply a value with a unique identity and a pointer to a predicate that serves as its class. (The predicate can be any procedure that returns true for that object, but is typically a message created with `make-class`—see section 4.11.) A **field** is a table of associations between values. Objects and fields are independent entities that can be tied together with branches that initialize field values when an object is created. Class inheritance can also be emulated with branches for a class predicate message that test an object for each subclass—in other words, a superclass predicate is the disjunction of its subclass predicates. Thus, new subclass relations can be added dynamically just by adding new branches.

The final primitive entity in Socrates is the **context**. Contexts can be used to return multiple values by name from a branch predicate to its body, such as intermediate values computed in the predicate or other auxiliary values that might be useful to the body. These values are returned as delayed expressions (closures), so that extra computation that is not needed when evaluating the predicate will only happen for those values that are retrieved from the context by the body. For example, a class predicate defined with `define-class` (see section 4.11) returns a context that provides access to a set of fields associated with the object, but each field accessor is only invoked if and when the field name is retrieved from the context.

In addition to these six core entities—messages, decision points, branches, objects, fields, and contexts—Socrates provides layers of syntactic sugar to emulate CLOS-style classes and multimethods [7], AspectJ-style pointcuts and advice [35], and ComposeJ-style composition filters [67]. It also supports all of the features of the predicate dispatching language defined

by Ernst, Kaplan, and Chambers [20], including predicate abstractions, structural pattern-matching, and classifiers.

The implementation of Socrates is embedded in PLT MzScheme [23] (an implementation of Scheme [34]) as a set of procedures and macros; thus, like MzScheme, 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. The complete source code for the implementation is available from SourceForge as the **socrates-lang** project [45].

1.3 Outline and research contributions

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 discusses the Socrates implementation; part IV compares Socrates to related work and explores some future directions in which this research could be extended.

The following are the main contributions of the research presented in this dissertation:

- The specification and implementation of a simple core programming language designed around support for separation of concerns.
- Syntactic sugar atop this core language to emulate the predicate dispatching language of Ernst, Kaplan, and Chambers [20]. Socrates is the most complete implementation of this language, including all the features described in [20] except for static type checking. Unlike their language, Socrates does not have a separate limited predicate language without recursion; predicates can be expressed using the full Socrates language.
- Syntactic sugar to emulate features from multiple aspect-oriented programming languages, including AspectJ and ComposeJ. This demonstrates the commonalities between these languages, and that they can be viewed as instantiations of predicate dispatching and open classes rather than orthogonal additions to object-oriented programming languages.

- A case study of a non-trivial GUI application implemented in Socrates. This provides evidence that Socrates is a practical language in which concerns such as view, model, selection, and error checking can be implemented in separate modules.
- A formulation of the Law of Demeter for Socrates, as well as a Socrates program to dynamically check for violations. This formulation synthesizes and extends previous formulations for CLOS and AspectJ.
- The specification and implementation of a partial-evaluator-based algorithm for analyzing predicate implication. This algorithm is potentially more efficient than the canonicalization algorithms described in the predicate dispatching literature [20, 11] because it can avoid processing unreachable subexpressions.

Part I

The Socrates programming language

Socrates is embedded in PLT MzScheme [23]; it is implemented as a collection of libraries that 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 [12].

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

The notation in the following chapters follows the same conventions as the MzScheme manual. Each procedure or syntactic form provided by Socrates is introduced with an expression pattern representing its possible uses. Plain, centered ellipses (\dots) indicate *zero* or more repetitions of the preceding pattern. Ellipses with a “1” superscript (\dots^1) indicate *one* or more repetitions of the preceding pattern. Square brackets (“[” and “]”) are used to designate optional arguments.

The naming conventions used by Socrates (and sample Socrates code) differ from typical Scheme naming conventions in two ways. The first difference is that the names of predicate procedures (including those representing classes—see section 4.11) always end in a question mark (?), even if they can return non-boolean values. The standard Scheme convention is that procedures that are ordinarily used as predicates but also return useful values, such as **memq** and **assoc**, do not have question marks in their names. Practically all predicates in a Socrates program, especially branch predicates (see section 2.3), return a context (see section 2.5) or something similar, and the question mark was kept to emphasize that they are meant to be predicates rather than being

undistinguished from other procedures or using a new naming convention. The second difference is that the names of some procedures and syntactic forms that modify global program state, such as `add-branch` and `define-method`, do not contain exclamation points. The standard Scheme convention is that the names of mutation procedures (“procedures that store values into previously allocated locations”, according to *R⁵RS* [34]) end in “!”. This convention is only used in Socrates when the procedure or syntactic form mutates its first argument, not when it indirectly mutates global storage. In addition, syntactic forms with the “`define-`” prefix are analogous to `define`, which does not have an exclamation point, even though (at the top level) it can cause mutation.

Chapter 2

Core procedures

This chapter describes the **socrates-core.ss** library. It provides six basic structures along with procedures to manipulate them (and one special form, **lambda/src**). Three of these structures are “static” values that make up a program: messages, branches, and fields. The other three are “dynamic” values that are created and used while running a program: decision points, objects, and contexts. All six entities are first-class Scheme values, however, so new messages, branches, and fields can also be created while running a program, i.e., not just when loading a module.

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 abstract operation; it can be sent to a tuple (list) of argument values to perform the operation on those values. The concrete 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 [7] 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 system state 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 the expression 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-all-previous dp`) returns a list containing `dp` and all previous decision points by repeatedly applying `dp-previous`.
- (`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 that 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, in an unspecified order.

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. These errors can be caught with MzScheme's exception handling mechanism; the `exn:msg:not-understood` and `exn:msg:ambiguous` structure types are both subtypes of `exn:msg`, which is a subtype of MzScheme's primitive exception type `exn:fail`. The decision point that caused the error can be extracted from any `exn:msg` instance using `exn:msg-dp`; also, `exn:msg:ambiguous-branches` extracts the list of equally-precedent branches from an `exn:msg:ambiguous` instance.

Because branch precedence is determined by inspecting the source expressions of predicate procedures, they should be created with the special form `lambda/src`, which is identical to `lambda` but creates a procedure that allows reflective access to the source expression and environment. (The implementation of `lambda/src` is described in section 7.1.) 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, while predicates may include side effects and may depend on global state, the predicate implication analysis makes some assumptions regarding these (see chapter 3) and care should be used when writing such predicates.

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. A filter procedure typically uses `append-contexts` to override context bindings returned from the next branch’s predicate (see section 2.5).

2.3.3 Branch precedence and around-branches

Branch precedence is determined by predicate implication analysis: if the predicate p_1 of a branch b_1 can be proven to logically imply 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 specifics of this analysis (as implemented in Socrates) are described in detail in chapter 3. For a simple example, consider the following two predicates:

```
(define p1
  (lambda/src (dp)
    (eq? (dp-msg dp) get-size)))

(define p2
  (lambda/src (dp)
    (and (eq? (dp-msg dp) get-size)
         (file? (car (dp-args dp))))))
```

Predicate `p2` implies predicate `p1`, because whenever an expression `(and e1 e2)` is true, the expression `e1` must also be true. Thus, a branch whose predicate is `p2` would have precedence over a branch whose predicate is `p1`.

Predicate implication is a generalization of the usual object-oriented method-overriding relation, in which subclass methods override superclass methods:

```
(define p3
  (lambda/src (dp)
    (and (eq? (dp-msg dp) get-size)
         (regular-file? (car (dp-args dp))))))
```

A branch whose predicate is `p3` would have precedence over a branch whose predicate is `p2`, because whenever the expression `(regular-file? e)` is true, `(file? e)` must also be true.

Sometimes a different precedence relation is needed:

```
(define (p4 dp)
  (and (eq? (dp-msg dp) get-bytes)
       (regular-file? (car (dp-args dp)))))

(define (p5 dp)
  (and (eq? (dp-msg dp) get-entry)
       (directory? (car (dp-args dp)))))

(define (p6 dp)
  (and (or (eq? (dp-msg dp) get-bytes)
          (eq? (dp-msg dp) get-entry))
       (file? (car (dp-args dp)))
       (not (get-readable? (car (dp-args dp))))))
```

The branch whose predicate is `p6` needs to have a higher precedence than the branches whose predicates are `p4` and `p5` so that it can raise an error for unreadable files, but `p6` does not imply either `p4` or `p5`. The implication 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 cause an ambiguous message error. (Giving the more recently defined around-branch precedence is somewhat arbitrary, but simple and unambiguous. This same precedence relation is used in the μ ABC calculus [8], which is compared with Socrates in section 9.3.1.)

2.3.4 Computing implication and precedence

Socrates provides procedures for analyzing predicate implication and branch precedence (e.g. in a reflective meta-predicate like that 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 branch-list)` returns a list of branches in the $branch-list$ that have the highest precedence—in other words, the minima of the set according to the partial order defined by `branch-precedes?`.

2.4 Fields and objects

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

An **object instance** is a unique value that can be used as a key in a field table. Each object has a pointer to its class.

- `(make-object class)` returns a new **object instance** whose class is $class$.
- `(object? v)` returns `#t` if v is an object instance, `#f` otherwise.
- `(object-class object)` returns the class of the object instance $object$.

An object's class can be any value, but by convention it is a predicate procedure that returns true for the object. This allows the predicate implication relation to serve as the superclass relation, with `object?` serving as the root class, since it returns true for all objects. One way to make such a class procedure is to use the recursive nature of Scheme's **define** statement:

```
(define regular-file?
  (lambda/src (x)
    (and (object? x)
         (eq? (object-class x) regular-file?))))
```

A superclass predicate can be defined to include its subclasses:

```
(define file?
  (lambda/src (x)
    (and (object? x)
         (memq (object-class x)
               (list file? regular-file? directory? ...))))))
```

Alternatively, a class predicate can be defined as a message and branch, so that the subclass cases of the predicate can be defined in separate branches without a class having to explicitly enumerate its subclasses. This is the approach taken by the **make-class** procedure (section 4.11).

The core Socrates language provides four standard messages, **alloc**, **init**, **make**, and **->list**, along with standard branches that provide default behavior for these operations. The first three messages form a generic construction protocol for instantiating objects and initializing field values:

- (**alloc** *class* *init-arg* ...) should return an object whose class is *class*. Typically a new object is created, but an existing object could be reused. The *init-arg* values may be used to determine how to allocate the object. The default method for **alloc** simply calls (**make-object** *class*).
- (**init** *class* *obj* *init-arg* ...) should initialize (by side-effect) the fields associated with object *obj* specific to the class *class* using the *init-arg* values as initialization arguments. Typically this will also send **init** to each of the superclasses of *class* to initialize the superclass fields associated with *obj*. The default method for **init** does nothing.
- (**make** *class* *init-arg* ...) should return an object whose class is *class* with its fields initialized using the *init-arg* values. The default method for **make** first calls (**alloc** *class* *init-arg* ...) to allocate an object *obj*, then calls (**init** *class* *obj* *init-arg* ...) to initialize *obj*, then returns *obj*.

A branch that handles the **init** message is known as an **initializer method**, and is roughly analogous to a constructor in other object-oriented languages. The initializer arguments may be any values; typically they are stored directly into fields based on their position in the list, but an initializer method could be defined to accept key-value pairs or any other protocol. Section 4.9 defines some higher-level syntax for defining initializer methods. For a typical class, no branches need to be added for the **alloc** and **make** messages, but some classes may require branches for custom allocation or object reuse.

Note that the construction protocol can be used for values other than objects; for example, (**make** *pair?* *x* *y*) could be made to return (**cons** *x* *y*) by adding an appropriate branch for

`alloc`. Non-object values returned by `make` should still be of the appropriate type, i.e. such that the `class` predicate procedure returns true.

The `->list` standard message provides a generic protocol for collecting field values of an object into a Scheme list:

- `(->list obj)` should return a list of the field values associated with the object `obj`. The default method for `->list` when sent to an object delegates to the object's class by calling `(->list (object-class obj) obj)`.
- `(->list class object)` should return a list of the field values associated with the value `v` for the class `class`. Typically this will also send `->list` to each of the superclasses of `class` and include those values in the returned list. The default method when sent to `object?` returns the empty list.

The order of the field values in the returned list typically corresponds to the order of the arguments expected by an initializer method for the object's class. This allows an object to be cloned using `(apply make (object-class obj) (->list obj))`.

2.5 Contexts

A **context instance** is a set of bindings of names (symbols) to expressions (closures). Contexts are typically returned from a branch predicate as a way to pass values to its body, such as intermediate values computed while evaluating the predicate or auxiliary values that might be useful to the body.

- `(make-context bindings-alist)` returns a new context instance from the association list `bindings-alist`, which contains 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* *name*) returns the value of the expression associated with the symbol *name* in the context instance *context* by invoking the corresponding thunk in the bindings association list. If *name* 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.

Chapter 3

Predicate implication analysis

When two plain branches or two around-branches are both applicable to a decision point, precedence between the two branches is determined by analysis of the implication relationship between their predicates: if one branch's predicate can be proven to logically imply the other's, but not vice versa, then the first branch precedes the second. Since a branch predicate can be any arbitrary Scheme procedure, this implication analysis is only an approximation, because it is undecidable in general whether one predicate implies another in a Turing-complete language. This chapter presents the analysis used by Socrates by sketching a simple algorithm to compute it; it is not the actual algorithm used by the Socrates implementation, which is more efficient but more complicated, but the two algorithms perform the same analysis. (The actual algorithm is described in chapter 8.)

The analysis makes two major simplifying assumptions about the predicates being compared. One is that no side effects in either of them affect either of their return values; side effect expressions (such as `set!`) are allowed in predicates, but they will be ignored by the implication analysis. The second is that free variables that are referred to by a predicate are assumed to remain constant over the lifetime of the predicate; their current values at the time of the analysis will be used in the analysis. (Most free variables in a predicate refer to other predicates, i.e. messages or Scheme procedures; replacing these with their current values can be thought of as a form of procedure inlining.)

If these assumptions hold, the analysis is conservative: 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 does not. This is analogous to static type systems that may reject type-correct programs that it cannot prove correct.

Predicate implication analysis happens at the time a message is sent, after two predicates have both been evaluated to true for a given decision point; the result of this analysis is memoized for all future comparisons of the same two predicates. However, these are both implementation details of the current version of Socrates and subject to change; for example, the analysis could be performed for all pairs of branches whenever a branch is added or removed.

The algorithm to determine whether a predicate procedure implies another has two stages:

1. Convert both predicate procedure body expressions into a canonical form consisting of atomic expressions combined with boolean operators **and** and **or**.
2. Use rules of boolean logic and knowledge of relationships between primitive procedures to determine whether one canonicalized expression implies another.

An **atomic expression** (or **atom**) is either a constant, a variable, an abstraction, or an application whose operator and operands are all atoms. A **primitive procedure** is any procedure that does not provide reflective access to its source expression and environment; this includes all of MzScheme's primitive procedures, but also all procedures created with **lambda** or **make-msg** instead of **lambda/src** or **make-msg/src** (see section 7).

3.1 Canonicalization

The procedure source expression stored by **lambda/src** has already been fully macro-expanded into primitive MzScheme syntax; in addition, all side-effect-only subexpressions have been eliminated (see section 7.1). The canonicalization algorithm thus applies to expressions consisting only of constants, variables, variable binding constructs (**let-values** and **letrec-values**), conditionals, applications, and abstractions.

An expression is canonicalized as follows:

1. Replace all free variables with their values as constants. Replace non-primitive procedure constants with their source abstraction expressions.
2. Convert variable binding expressions into applications of abstractions (or `call-with-values`).
3. Reduce all applications of abstractions by substituting argument expressions for formal variables in the abstraction body.
4. Lift all conditionals out of applications and conditional tests using these distribution rules:
 - $(e1 \dots (\mathbf{if} \ e2 \ e3 \ e4) \ e5 \dots) = (\mathbf{if} \ e2 \ (e1 \dots e3 \ e5 \dots) \ (e1 \dots e4 \ e5 \dots))$
 - $(\mathbf{if} \ (\mathbf{if} \ e1 \ e2 \ e3) \ e4 \ e5) = (\mathbf{if} \ e1 \ (\mathbf{if} \ e2 \ e4 \ e5) \ (\mathbf{if} \ e3 \ e4 \ e5))$
5. Convert conditionals into boolean formulas using the following rule:
 - $(\mathbf{if} \ e1 \ e2 \ e3) = (\mathbf{or} \ (\mathbf{and} \ e1 \ e2) \ (\mathbf{and} \ (\mathbf{not} \ e1) \ e3))$

Steps 1 and 3 are applied repeatedly to their results, except when a variable's value is a procedure already being replaced or reduced (which would cause an infinite loop). Bound variables are renamed as necessary to avoid capture.

Atoms are further simplified by reducing applications that compose complementary primitive procedures. For example:

- $(\mathbf{car} \ (\mathbf{cons} \ e1 \ e2)) = e1.$
- $(\mathbf{apply} \ op \ e1 \dots (\mathbf{list} \ e2 \dots)) = (op \ e1 \dots e2 \dots).$
- $(\mathbf{assq} \ e1 \ (\mathbf{list} \ (\mathbf{cons} \ e1 \ e2) \ e3 \dots)) = e2.$
- $(\mathbf{not} \ (\mathbf{not} \ e)) = e.$

Applications of primitive procedures to constants are also reduced. See section 8.8 for more details about primitive application.

3.2 Implication of canonicalized expressions

The decision of whether a canonicalized expression implies another is based on the following rules:

- (**and** $e1$ $e2$) implies both $e1$ and $e2$.
- $e1$ and $e2$ both imply (**or** $e1$ $e2$).
- $e1$ and $e2$ imply each other if they are syntactically equivalent (see section 3.5).
- $e1$ implies $e2$ if $e1$ is a false constant or $e2$ is a true constant, an abstraction, or an application of a tautology (see section 3.4).
- (**p1** $e1$ \dots) implies (**p2** $e2$ \dots) if ($e1$ \dots) and ($e2$ \dots) are syntactically equivalent, $p1$ and $p2$ are primitive procedures, and $p1$ implies $p2$ (see section 3.3). For example, (**integer?** x) implies (**number?** x), and ($=$ x y) implies ($<=$ x y).
- (p $e1$) implies $e2$ if $e1$ and $e2$ are syntactically equivalent and p is a primitive procedure that returns false when applied to false. For example, (**integer?** x) implies x , because (**integer?** **#f**) is false.
- (**eq?** $e1$ $e2$) implies $e3$ if substituting $e1$ for $e2$ (or vice versa) in $e3$ produces an expression that can be reduced to a true constant or an abstraction. For example, (**eq?** x 3) implies (**integer?** x) because (**integer?** 3) is true.
- (**not** $e1$) implies (**not** $e2$) if $e2$ implies $e1$.
- $e1$ implies (**not** $e2$) if $e1$ and $e2$ are disjoint atoms.

Two atoms are **disjoint** if they are never both true or both false. This is decided based on the following rules:

- (**p1** $e1$ \dots) and (**p2** $e2$ \dots) are disjoint if ($e1$ \dots) and ($e2$ \dots) are syntactically equivalent and $p1$ and $p2$ are disjoint primitive procedures (see section 3.3). For example, (**integer?** x) and (**string?** x) are disjoint, as are ($<$ x y) and ($>$ x y).

- `(eq? e1 e2)` is disjoint from `e3` if substituting `e1` for `e2` (or vice versa) in `e3` produces an expression that can be reduced to a false constant. For example, `(eq? x 3)` implies `(string? x)` because `(string? 3)` is false.

3.3 Primitive implication and disjointness

Implication and disjointness relations between primitive procedures can be asserted with the following procedures from the *prim-implication.ss* module:

- `(add-implication! prim1 prim2)` asserts that *prim1* directly implies *prim2*. Implication is the transitive closure of the direct implication relation. (Cycles are permitted; primitives in a cycle are considered logically equivalent.)
- `(add-disjoint-set! prim-list)` asserts that all of the primitives in *prim-list* are pairwise directly disjoint. Disjointness is the composition of the direct disjoint relation and implication; in other words, two primitives are disjoint if they are directly disjoint or they imply primitives that are directly disjoint.

Syntactic sugar is also provided:

- `(declare-prim-implies prim0 prim ...)` asserts that *prim0* directly implies each *prim*.
- `(declare-disjoint prim ...)` is equivalent to `(add-disjoint-set! (list prim ...))`.
- `(declare-partition prim0 prim ...)` asserts that each *prim* directly implies *prim0* and that all *prims* are pairwise directly disjoint.

The above three forms can appear in either a definition lexical context or an expression lexical context.

The following primitive relationships are declared in Socrates by default:

```
(declare-disjoint boolean? pair? symbol? number? char? string?
                 vector? port? procedure? null?)
```

```

    hash-table? promise?)
(declare-prim-implies null? list?)
(declare-partition char? char-alphabetic? char-numeric? char-whitespace?)
(declare-partition char-alphabetic? char-upper-case? char-lower-case?)
(declare-prim-implies complex? number?)
(declare-prim-implies real? complex?)
(declare-prim-implies rational? real?)
(declare-prim-implies integer? rational?)
(declare-partition integer? even? odd?)
(declare-partition number? exact? inexact?)
(declare-partition number? positive? negative? zero?)
(declare-prim-implies zero? even?)
(declare-partition port? input-port? output-port?)

(declare-prim-implies boolean=? eq?)
(declare-prim-implies eq? eqv?)
(declare-prim-implies char=? eqv?)
(declare-prim-implies eqv? equal?)
(declare-prim-implies string=? equal?)
(declare-prim-implies equal? =)
(declare-partition <= < =)
(declare-partition >= > =)
(declare-disjoint <= >)
(declare-disjoint >= <)

```

3.4 Primitive tautologies

Some primitives are tautologies, i.e. always return true values:

- `(prim-tautology? v)` returns `#t` if `v` is a structure constructor procedure or a primitive that is known to always return true, such as `cons` or `+`, or `#f` otherwise.
- `(add-prim-tautology! prim)` adds `prim` to the list of primitives that always return true.

3.5 Syntactic equivalence

Two canonicalized expressions are **syntactically equivalent** if they have the same structure, corresponding constants are `equal?`, and corresponding variables refer to the same bindings. Since all free variables and variable binding constructs have been removed, this means that variables must refer to the same formal argument position in the same enclosing abstraction. In other words, corresponding variables must have the same **lexical address**. Computing lexical addresses is a well-known algorithm (e.g. [2]).

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 lexical context, similar to how MzScheme infers procedure names, using MzScheme's **syntax-local-name** procedure. (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); otherwise, it is equivalent to `(begin)`, i.e., the null statement. 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)
```

`(unparse-dp dp)` returns a list representing the application expression that created the decision point `dp`: the name of the message followed by the arguments. If `dp` is false, the string `"<top-level>"` is returned.

The module `print-handler.ss` provides the `print-handler` message, and installs it as the global port print handler (the default printer is provided as `default-global-port-print-handler`). `(print-handler v)` prints `v` to the current output port:

- If `v` is a message, it is printed as `#<msg:name>` where `name` is the name of the message (as opposed to `#<procedure:name>` as the default printer prints).
- If `v` is an object whose class is a message, it is printed as `#<object:name>` where `name` is the name of the message (as opposed to `#<struct:object>` as the default printer prints). If `v` is an object whose class is not a message, it is printed as `#<object>`.
- If `v` is a pair or vector, it is printed as normal, but the contents are printed using `print-handler`.

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*. (The underlying procedure is renamed to **make-context***.) It is equivalent to the following expression:

```
(make-context* (list (cons 'name (lambda () expr)) ...))
```

(lambda-context (*name* ...) *body* ...) 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. This is typically used to define a branch body, to make some of the context bindings from the branch predicate available as local variables. For example:

```
(define regular-file?
  (lambda/src (x)
    (and (object? x)
         (eq? (object-class x) regular-file?)
         (make-context (inode (get-inode x))
                       (contents (get-contents x))
                       (size (get-size x))))))

(define-msg stat)

(define-branch
  (lambda/src (dp)
    (and (eq? (dp-msg dp) stat)
         (regular-file? (car (dp-args dp)))))
  (lambda-context (inode size)
    (printf "Inode: ~a Size: ~a" inode size)))
```

(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. It 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 name-binding and pattern-matching similar to that of the predicate dispatching language [20].

`(return (name expr) ...)` is a synonym for `(make-context (name expr) ...)`.

`(and:= bind-expr ...)` is an extension of Scheme's short-cutting `and` form to allow variables to be bound during the evaluation of an expression and referenced in later expressions. The syntax of binding expressions is as follows:

bind-expr is one of

```
expr
(:= variable expr)
(or:= bind-expr ...)
(and:= bind-expr ...)
(=> expr (bind-pattern ...1)
```

The `:=` form binds the variable to the result of evaluating the expression; this binding is available to the remaining binding expressions in the enclosing `and:=` expressions. In other words,

```
(and:= (:= variable expr) bind-expr ...)
```

is equivalent to

```
(let ((variable expr)) (and:= bind-expr ...))
```

For example, the following predicate tests whether x is a directory containing a file named n , and if so returns a context mapping the name $file$ to the file:

```
(define directory-containing-file?
  (lambda/src (x n)
    (and:= (directory? x)
           (:= f (get-entry x n))
           f
           (return (file f))))))
```

The **or:=** form extends Scheme's short-cutting **or** form to allow binding expressions: the bindings from the first expression that evaluates to true are available to the remaining binding expressions in the enclosing **and:=** expressions.

The \Rightarrow form allows variables to be bound by pattern matching on contexts (see section 2.5). If the expression evaluates to a true value (which must be a context), the binding patterns will be evaluated using the values of the corresponding context expressions. The syntax of binding patterns is as follows:

```
bind-pattern is one of
  (:= variable variable)
  (expr variable expr ...)
  (expr (:= variable variable) expr ...)
  ( $\Rightarrow$  (expr variable expr ...) bind-pattern ...1)
  ( $\Rightarrow$  (expr (:= variable variable) expr ...) bind-pattern ...1)
```

The **:=** binding pattern binds the first variable to the value of the context expression corresponding to the second variable. This binding is available to all the other binding patterns in this \Rightarrow form as well as the remaining binding expressions in the enclosing **and:=** forms.

A binding pattern of the form (*expr variable expr* ...) acts as a specializer test: it is evaluated as an application expression, with the context expression corresponding to the variable used as the first argument; the test must evaluate to true for the remaining binding expressions to be evaluated. The (*expr* (:= *variable variable*) *expr* ...) form is similar, but also binds the first variable to the value of the context expression corresponding to the second variable for

use outside this test. A `=>` binding pattern can be used to match further binding patterns on the context instance returned by the expression, optionally binding the context expression to a variable.

For example, the following predicate tests whether `x` is a directory containing a regular file named `n` whose size is no bigger than `m`, and if so returns a context mapping the names `file` and `size` to the file and its size:

```
(define directory-containing-small-regular-file?
  (lambda/src (x n m)
    (and:= (= > (directory-containing-file? x n)
              (= > (regular-file? (: f file))
                  (<= (: s size) m)))
           (return (file f) (size s))))))
```

This `and:=` expression is equivalent to the following:

```
(let ((c1 (directory-containing-file? x n)))
  (and c1
       (let* ((f (context-value c1 'file))
              (c2 (regular-file? f)))
         (and c2
              (let ((s (context-value c2 'size)))
                (and (<= s m)
                     (make-context (file f) (size s))))))))))
```

Other examples of binding patterns appear in sections 5.3 and 9.1.1.

`(predicate formal-patterns [bind-expr])` returns a new procedure (using `lambda/src`). The `formal-patterns` form is similar to the `formals` form of a `lambda` expression, but each argument position can be a pattern to be matched instead of just a variable. The syntax is determined by the following grammar:

```
formal-patterns is one of
  variable
  (formal-pattern ...)
```

(formal-pattern ...¹. variable)

formal-pattern is one of

variable

(expr variable expr ...)

(=> (expr variable expr ...) bind-pattern ...)

A formal pattern is similar to a binding pattern, but the variable is bound to the corresponding argument value by position, rather than to the value of a context expression retrieved by name. The body of the predicate procedure is an **and:=** expression containing all the tests and bindings from the formal patterns, as well as *bind-expr* (if supplied). In particular, this means that the variables bound by the *formal-patterns* are available in *bind-expr*.

Some useful predicate procedures 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.

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 [35].

(**call** *msg* \dots^1) is equivalent to (**predicate** (*dp*) (**or** (**eq?** (**dp-msg** *dp*) *msg*) \dots^1)).

(**args** *pred-expr* \dots) 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 instance 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.


```
(follow-next-branch
  (body-expr context)))
```

(The syntactic form **begin0** in MzScheme is like **begin** but its values are the values of the first expression instead of the last.)

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

```
(make-around pred-expr (lambda/src (context)
  (call-with-values (lambda () (follow-next-branch))
    (lambda vals
      (apply (lambda vars (body-expr context)) vals)
      (apply values vals))))))
```

(**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 [7] or the predicate dispatching language [20].

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

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

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

```
(make-branch (method-call (msg . formal-patterns) [bind-expr])
  (lambda-context (var ...) body-expr ...1))
```

where the *vars* are the bound variables in the *formal-patterns* and *bind-expr* forms. 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.

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

```
(begin (ensure-msg name)
      (add-branch
       (make-method (name . formal-patterns) [& bind-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 syntactic sugar for field accessor branches.

The Socrates naming convention for field 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))
```

¹MzScheme already has a **when** syntactic form, and overloading it would be confusing.

(**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 Initializer sugar

The module **init-sugar.ss** provides some syntactic sugar for defining initializer methods.

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

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

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

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

```
(add-branch (make-init-method ((type-expr this) . formal-patterns) [& bind-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.

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*) [*bind-expr*]) is similar to **(method-call (*msg . formal-patterns*) [*bind-expr*])** except that instead of returning a context instance that binds the variables in *formal-patterns* and *bind-expr*, it returns the value of *bind-expr* if it is 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*) [*bind-expr*]) is equivalent to

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

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

```
(begin
  (define-predicate (name . formal-patterns)
    (and:= [common-bind-expr] bind-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-bind-expr*] (*name bind-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. The keyword **otherwise** can be used in place of the final *bind-expr*, which indicates a default classification if all the other predicates are false.

(**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 binding each *field-name* to an expression that sends that field's getter message to the object; in other words, the field is only retrieved if and when the field name is retrieved from the context. 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.

For example, (**define-class** `directory? (file?) (entries)`) is equivalent to the following:

```
(define directory? (make-class (lambda/src (x)
                                (return (entries (get-entries x))))))
(declare-implies directory? file?)
(define-field entries directory?)
(define-init-method ((directory? x) entries . super-inits)
  (apply init file? x super-inits)
  (set-entries! x entries))
(define-method (->list (eq? class directory?)
                    (=> (directory? x) (:= entries entries)))
  (append (list entries) (->list file? x)))
```

4.12 Composition filters sugar

The module **filter-sugar.ss** provides syntactic sugar to emulate **composition filters** [3, 6]. 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.

²A custom initializer method must be defined for a class with multiple supertypes requiring initialization 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 *filter* in sequence. (`define-output-filters type filter ...`) is similar, but the attached filters process decision points when *type* is true for the sender. In both cases, the filters ignore decision points satisfying any of the following three helper predicates:

- (`self-call? dp`) returns true if the receiver of *dp* is the same as the sender.
- (`class-call? dp`) returns true if the sender of *dp* is a procedure that returns true when applied to the receiver. This is analogous to a `static` method (or constructor) in Java calling a method on an object of the method's class; the convention in Socrates (e.g., for the `init` message protocol; see section 2.4) to emulate a `static` method is to use the class as the first argument.
- (`pred-call? dp`) returns true if the message of *dp* is a predicate message (according to `msg/src?`; see section 7.2). These are ignored in order to avoid infinite recursion while evaluating a branch predicate.

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 [3] 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

The file system example in figures 3 and 4 demonstrated some of the expressiveness of the syntactic sugar in the full Socrates language. The following two chapters expand on this by describing two non-trivial programming projects implemented in Socrates. In particular, they demonstrate how Socrates supports separation of concerns by allowing a combination of the predicate dispatching and aspect-oriented styles of programming.

Chapter 5 describes a domino puzzle GUI applet, with separate sets of modules for the model and view concerns as well as for error handling and maintaining the current selection. Chapter 6 defines the Law of Demeter for Socrates and describes a dynamic checker that can be installed simply by loading its module along with the code to be checked.

Chapter 5

Domino puzzle

The domino puzzle project was inspired by a pencil-and-paper 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 numbers on the domino. A domino puzzle applet with a graphical user interface was implemented in Socrates using PLT Scheme’s MrEd GUI toolkit [25]. Figure 5 shows a screenshot. A randomly generated¹ board is to the left, with unplaced dominos to the right. Each domino has two numbers between 1 and 6. Clicking on a number on a domino (on or off the board) will select the domino. The selected number is highlighted red. Selecting a domino off the board enables the “rotate left” and “rotate right” buttons, which can be used to rotate the selected domino by 90 degrees counterclockwise or clockwise, respectively. Clicking an uncovered number on the board will move the selected domino to the board (assuming the numbers match). Selecting a domino on the board and then clicking off the board will remove the domino from the board; this is better than having to undo mistakes with a pencil eraser!

The two main concerns to be separated are 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.

¹Only solvable boards are generated, although the solution is not guaranteed to be unique.


```

(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 6: Class definitions for the domino-puzzle model. The field names indicate the intended types of the field values, but they are not checked by the default initializer methods generated by `define-class`.

5.1 The model concern

The top-level class in the model is `polyomino-puzzle`. (The implementation generalizes 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, i.e. a two-dimensional array of cells that each contain a tile or `#f`². 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° 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 a `polyomino` could only be placed 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 6.

The main operation for playing the puzzle is implemented by the `place!` method, which places a `polyomino` onto a `board` at a given position by copying the tiles from its tile matrix to the board’s tile matrix:

```

;; Place a polyomino on the board with its origin at the given position.
(define-method (place! (board? x) (position? pos) (polyomino? p))
  (for-each-tile p
    (lambda (tpos t)
      (set-tile! x (offset pos (offset-from-origin tpos)) t))))

```

²The occupied cells of a polyomino tile matrix do not need to form a solid rectangle—they need not even be contiguous—but the random board generator only works if this is true.

```

(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 (message (string-append "Off the board: " (->string p))))))
  (already-occupied?
    (and (occupied? x p)
      (return (message (string-append "Already occupied: " (->string p))))))
  (label-does-not-match?
    (let ((l1 (get-label x p)) (l2 (get-label t)))
      (and (labels-do-not-match? l1 l2)
        (return (message (format "Doesn't match: ~a ~a" l1 l2))))))

```

Figure 7: Branches of the `bad-tile-placement?` predicate.

Position is an auxiliary class from the matrix module that encapsulates x and y coordinates. The `for-each-tile` method invokes a procedure taking two arguments, a position and a tile, for each tile in the tile matrix of a polyomino. The `offset` and `offset-from-origin` methods from the matrix module are used to convert the position of the tile in the polyomino tile matrix to a position in the board tile matrix.

The error-checking for the `place!` operation is separated into its own branch:³

```

(define-method (place! (board? x) (position? pos) (polyomino? p))
  & (=> (bad-placement? x pos p) (:= msg message))
  (error `place! msg))

```

The `bad-placement?` predicate maps `bad-tile-placement?` over each tile in the polyomino, returning a context instance that binds `message` to an expression that evaluates to an appropriate error message (which is bound to `msg` in the above method). The latter predicate is specified with the help of a classifier (see figure 7).

³Note that the third line of the method definition is indented to line up with the ampersand on the second line, rather than the open parenthesis that begins the predicate expression. This keeps it clear that the third line belongs to the branch body, not the 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 8: The view-model association.

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 8. There is also a type constructor `viewed` that distinguishes model objects that have views attached from ones that do not; 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 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 widget⁴:

```

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

```

Because the GUI does not use drag-and-drop⁵, placing polyominos onto the board is a two-step

⁴The `send` special form is from MrEd's object system, which is distinct from Socrates's; in particular, it is 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).

⁵The MrEd toolkit does not support drag-and-drop in a straightforward way, requiring the use of either OpenGL or the Editor Toolbox, two large and complicated subsystems.

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. This is 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 does not have these backlinks, because it does not 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 9). 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 information, we can also define some predicates that partition tiles, polyominos, and polyomino-puzzles into subtypes (see figure 10), and use these predicates to define error-checking branches separately from the main operations (see figure 11).

5.4 Analysis

The main goal of the domino puzzle program is to demonstrate the expressiveness of the Socrates syntax and its ability to emulate the mechanisms supporting separation of concerns in other languages. However, it is also interesting to examine some quantitative measurements. The program is small but non-trivial: 26 modules containing a total of 1455 lines of code. When the program is loaded, there are 173 messages and 410 active branches (including those provided by

```

(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-polyomino-set! (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 9: Fields and advice to maintain backlinks for the model containment hierarchy.

```

(define-predicate (tile-on-board? (tile? x))
  (let ((p (get-polyomino x)))
    (and:= (=> (polyomino-on-board? p) (:= b board))
            (return (polyomino p) (board b))))))
(define-predicate (tile-off-board? (tile? x))
  (let ((p (get-polyomino x)))
    (and:= (=> (polyomino-off-board? p) (:= ps polyomino-set))
            (return (polyomino-set ps))))))

(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 ((=> (polyomino-puzzle? x) (:= ps polyomino-set) (:= b board))
  (puzzle-with-selection?
    (let ((t (get-selected-tile x)))
      (and t
            (let ((p (get-polyomino t)))
              (return (polyomino-set ps) (board b) (selected-tile t) (selected-polyomino p)
                      (selected-polyomino-parent (get-parent p)))))))
  (puzzle-with-no-selection?
    (return (polyomino-set ps) (board b))))))

(define-predicates ((=> (puzzle-with-selection? x) (:= ps polyomino-set) (:= b board)
                       (:= t selected-tile) (:= p selected-polyomino)
                       (:= pp selected-polyomino-parent))
  (puzzle-with-selection-on-board?
    (and (eq? pp b)
          (return (polyomino-set ps) (board b) (selected-tile t) (selected-polyomino p)
                  (selected-polyomino-position (get-position p))))))
  (puzzle-with-selection-off-board?
    (and (eq? pp ps)
          (return (polyomino-set ps) (board b) (selected-tile t) (selected-polyomino p))))))

```

Figure 10: 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 11: Error checking for polyomino-puzzle operations.

Socrates itself). While the program is a good example of separation of concerns (into many small modules containing many small branches), it is not necessarily a good example of crosscutting concerns: there are only 17 around-branches, and only 5 branches that are applicable to multiple messages (two of these are shown in figure 11). There are 101 messages for which more than one branch is applicable, however, which demonstrates a high degree of polymorphism.

A typical run through the domino puzzle GUI (performing all the operations and producing each possible error message) results in 274 unique sets of applicable branches. Of these, 84 contain two branches, 13 contain three branches, one contains four branches, and one contains five (shown in figure 12). In other words, there are 99 different cases where sending a message results in the branch precedence algorithm being invoked. These represent a combined 69,020 messages sent, out of a total of 175,528, or roughly 39%; however, since the results of the precedence algorithm are cached, the actual precedence computation only occurs once for each of the 99 sets. More precisely, precedence is computed once for each of the $84 \cdot 1 + 13 \cdot 3 + 1 \cdot 6 + 1 \cdot 10 = 139$ different pairs of branches that are compared. Of these, 41 pairs involve one around-branch and one plain branch; in other words, predicate implication analysis is performed 98 times.

The performance of the domino puzzle implementation is reasonable: on a 5-year-old computer with an 800 MHz Athlon chip running Debian Linux, there is a small (2–4 second) pause the first time each new operation is performed (due to implication analysis) but the GUI is otherwise quite responsive. Creating and displaying a domino puzzle (from a fixed number grid, i.e. without random board generation) takes roughly 18 seconds of CPU time the first time and roughly

```

;; Catch-all method for non-rotatable labels.
(define-method (rotate! x n) (void))

;; Rotate a polyomino by rotating its tile matrix (and all its cells).
(define-method (rotate! (polyomino? x) (integer? n))
  (rotate! (get-tile-matrix x) n))

;; Update the backlinks after modifying a polyomino's tile matrix.
(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))))))

;; Replace the view of a polyomino with a new one after rotating it.
(define-after-method (rotate! (=> ((viewed polyomino-off-board?) x)
  (:= ps polyomino-set))
  (integer? n))
  (let ((psv (get-view ps)))
    (set-child! psv (list-index (eq x) (get-polyominos ps))
      (make-view x psv #t))))

;; Re-highlight the selected tile after rotating its polyomino.
(define-after-method (rotate!
  (=> ((viewed selected-polyomino-off-board?) x)
  (:= tile selected-tile))
  (integer? n))
  (highlight tile))

```

Figure 12: Five overlapping branches, in order of ascending precedence. (This is the order in which the bodies of the last four are invoked, since the last three are after-methods.)

12 seconds each time thereafter. About half this time is spent in the garbage collector; this might be due to the weak hash tables that are used to cache the results of predicate implication analysis (see section 8.9), or it may just be that the domino puzzle implementation (or Socrates itself) is generating too many intermediate data structures. See section 10.4 for more discussion of the efficiency of Socrates.

Chapter 6

Law of Demeter checker

The Law of Demeter [41, 39, 38] is a style rule for object-oriented design. In order to reduce dependencies between classes (or modules), a method should only send messages to a restricted set of closely-related objects; the motto is “don’t talk to strangers”. Aspect-oriented programming makes it possible to write a run-time checker for violations of the Law of Demeter [37]. This chapter provides a formulation of the Law of Demeter for Socrates and presents an implementation of a run-time checker in Socrates.

6.1 The Law of Demeter for Socrates

There have been many formulations of the Law of Demeter for different object-oriented models and languages. The following definitions state the Law of Demeter for a single-dispatch object-oriented model such as UML [54]:

Definition 1 A *supplier* to a method M is an object to which a message is sent in M . The *potential preferred suppliers* to a method M are all of the following:

- the argument objects of M (including self)

- *attributes of self and objects associated with self*
- *objects created during the execution of M*
- *objects in global variables*

An object is a **preferred supplier** to a method *M* if it is both a supplier and a potential preferred supplier to *M*. **Law of Demeter:** Every supplier to a method *M* must be a preferred supplier to *M*.

Attributes and associations can be stored in fields or computed by methods; this means that any object returned from sending a message to *self* is a potential preferred supplier. Global variables include class names, so invoking class methods is always allowed.

The Law of Demeter for CLOS [40] extends the definitions for a multiple-dispatch model:

Definition 2 A **method selection argument** of a generic function call is an argument that is used for identifying the applicable methods. A **supplier** object to a method *M* is a method selection argument of a function call in *M*. The **potential preferred supplier** objects to a method *M* with method selection argument objects *A* are all of the following:

- *the argument objects of M*
- *slot values of objects in A*
- *return values of function calls whose method selection argument objects are all in A*
- *objects created during the execution of M*
- *objects in global variables*

The Law of Demeter over Join Points [37] is a reformulation for AspectJ's join point model:

Definition 3 The **potential preferred supplier** objects to a join point *J*, child of the enclosing join point *E*, are all of the following:

- the argument objects of E
- return values of the siblings of J which do not have a target (such as constructor calls) or whose target is the target of E

Law of Demeter over Join Points: For each join point J , the target of J must be a potential preferred supplier to J .

These can be synthesized into a Law of Demeter for Socrates:

Definition 4 A **decision value** of a decision point D is a value used to select applicable branches for D . (This is a generalization of the method selection argument concept, since other values besides arguments may influence branch selection.) An **associated value** of a decision point D is a return value of a decision point whose previous decision point is D and whose decision values are all decision values of D . The **source procedure** of a decision point D is the body procedure of the branch being followed when D was created. The **potential preferred suppliers** to a decision point D with source procedure S and previous decision point P are all of the following:

- the arguments of P
- associated values of P
- objects created during the processing of P
- the values of the free variables of S

The Law of Demeter for Socrates: Every decision value of a decision point D must be a potential preferred supplier to D .

6.2 Checking the Law of Demeter in Socrates

Figure 13 shows the code for a predicate to detect decision points that violate the Law of Demeter. It follows the definition fairly directly, with two exceptions:

```

(define (lod-violation? dp)
  (let ((ill (illegal-suppliers dp)))
    (and (not (null? ill))
         (return (dp dp) (ill ill))))))

(define (illegal-suppliers dp)
  (lset-difference eq?
    (filter object? (decision-values dp))
    (potential-preferred-suppliers dp)))

(define (decision-values dp)
  (dp-args dp))

(define (potential-preferred-suppliers dp)
  (let ((prev (dp-previous dp)))
    (append (dp-args prev)
            (get-associated-values prev)
            (get-created-objects prev)
            (free-var-values (source-proc dp)))))

(define (source-proc dp)
  (branch-body (dp-within dp)))

(define (free-var-values proc)
  (map (lambda (var) (proc-env-value proc var))
       (proc-env-vars proc)))

```

Figure 13: A decision point predicate for Law of Demeter violations.

- Non-object values are always considered preferred suppliers, since they are not easily trackable—they might be constructed by primitives or literal expressions. This could be considered a generalization of allowing free variables and created objects.
- Since determining the decision values of a decision point would require analyzing the predicate expressions of all applicable branches, the set of decision values is approximated by using the set of arguments.

The associated values of a decision point are determined by recording them in a field using an after-branch that detects uses of a previous decision point's decision values (figure 14). Two more approximations are used in this implementation: the elements of an associated list value of a decision point are considered to be associated values, and the elements of each list argument of a decision point are considered to be associated values. This emulates the idea of one-to-many associations without requiring the use of special collection classes.

```

(define-field associated-value-list dp? null)

(define-method (add-associated-value! (dp? dp) (object? v))
  (set-associated-value-list! dp (cons v (get-associated-value-list dp))))

(define-method (add-associated-value! (dp? dp) (list? l))
  (for-each (lambda (v) (add-associated-value! dp v)) l))

(define-method (add-associated-value! (dp? dp) v)
  ;; Non-objects are always legal suppliers, don't bother
  ;; remembering them as associated values.
  (void))

(define-method (get-associated-values (dp? dp))
  (append
   (concatenate (filter (list-of object?) (dp-args dp)))
   (get-associated-value-list dp)))

(define (use-of-prev-dec-vals? dp)
  (and (dp-previous dp)
       (let ((dec-vals (decision-values dp)))
         (and (not (null? dec-vals))
              (subset? dec-vals (decision-values (dp-previous dp)))))))

(define-after (&& use-of-prev-dec-vals?
              (! (cflow (call add-associated-value!))))
  returning (val)
  (lambda-context ()
    (add-associated-value! (dp-previous (current-dp)) val)))

```

Figure 14: Tracking associated values.

The created objects of a decision point are similarly determined by an after-branch that records the return value of an `alloc` message in a field on each previous decision point on the stack (figure 15). This relies on usage of the construction protocol defined in section 2.4 rather than calling `make-object` directly.

Figure 16 shows the before-branch that detects Law of Demeter violations and the methods for reporting them. Figure 17 shows an example of some non-violations, a violation, and the checker output.

```

(define-field created-objects dp? null)
(define-method (add-created-object! (dp? dp) (object? obj))
  (set-created-objects! dp (cons obj (get-created-objects dp))))
(define-after (call alloc) returning (obj)
  (lambda-context ()
    (for-each (lambda (prev) (add-created-object! prev obj))
      (dp-all-previous (current-dp)))))

```

Figure 15: Tracking created objects.

```

(define-before
  (&& (cflowbelow ?) ;not at top-level
    (! (args dp? ..)) ;allow predicates and reflective methods
    (! (cflow (call print-handler))) ;allow printing
    (! (cflow (call report-lod-violation))) ;recursion guard
    lod-violation?)
  (lambda-context (dp ill)
    (report-lod-violation dp ill)))
(define-method (report-lod-violation (dp? dp) (list? ill))
  (report-lod-violation-dp dp)
  (report-lod-violation-ill ill))
(define-method (report-lod-violation-dp (dp? dp))
  (printf "~%Law of Demeter violation!~%"
    (printf "Branch: ")
    (browse-branch (dp-within dp))
    (printf "Previous: ~v~%" (unparse-dp (dp-previous dp)))
    (printf "Message: ~v~%" (unparse-dp dp))))
(define-method (report-lod-violation-ill (singleton-list? ill))
  (printf "Illegal supplier: ~v~%" (car ill)))
(define-method (report-lod-violation-ill ill)
  (printf "Illegal suppliers: ~v~%" ill))

```

Figure 16: Detecting and reporting violations.

```

(define-class a? () (b))
(define-class b? () (c))
(define-class c? () (d))
(define-class d? () ())

(define the-a #f)

(define-method (violate-lod)
  (set! the-a (make a? (make b? (make c? (make d?))))))
  (violate-lod (make a? (get-b the-a))))

(define-method (violate-lod (a? an-a))
  ;; Not a violation: sending a message to an argument
  (get-b an-a)
  ;; Not a violation: sending a message to an argument's associated value
  (get-c (get-b an-a))
  ;; Not a violation: sending a message to a locally created object
  (get-b (make a? (get-b an-a)))
  ;; Not a violation: sending a message to a free variable
  (get-b the-a)
  ;; A violation:
  (get-d (get-c (get-b an-a)))
  (void))

Law of Demeter violation!
Branch: (method-call (violate-lod (a? an-a)))
Previous: (violate-lod #<object:a?>)
Message: (get-d #<object:c?>)
Illegal supplier: #<object:c?>

```

Figure 17: Examples and checker output.

6.3 Checking the domino puzzle code

As a test, the Law of Demeter checker was used to check the domino puzzle program from chapter 5. This uncovered several bugs and limitations in the Socrates implementation, mostly related to predicate implication analysis. For example, in order to determine the source expression for a predicate message, all branches that might ever apply to that message must be considered; this includes the around branches from the Law of Demeter checker, so a partial procedure (see section 8.8) had to be defined to expand a call to **follow-next-branch** into the predicate and body of the next branch. It also uncovered some problems with the Law of Demeter checker implementation, leading to the approximations described in section 6.2.

Once these issues were addressed, the domino puzzle GUI could be run successfully, and a number of Law of Demeter violations were reported. For example, the predicate method **has-neighbor?** used when drawing the border of a polyomino (figure 18) contains a violation:

- The value of x is a preferred supplier, because it is an argument.
- The value of `(get-polyomino x)` is also a preferred supplier, because it is an associated value.
- The value of `(get-position (get-polyomino x))` is not a preferred supplier, however, so sending `in-range?` to it is a violation.

This could be avoided by factoring the last two lines into a new predicate method **has-tile?** (figure 19). This refactoring is a good idea anyway, because it separates the details of the interface of the tile class (the `get-position` and `get-polyomino` methods) from the details of the interface of the polyomino class (the `get-dimensions` and `get-tile` methods). The Law of Demeter for Socrates checker enforces the information hiding principle, which has long been known to be an effective criteria for module decomposition [51], without requiring static type and interface declarations.

A few false positives were also reported, due to objects being returned from predicates in context instances. For example, a violation is reported in the after-method that highlights a selected tile after its polyomino is rotated (figure 20), because the tile is an associated value of the

```

(define-predicate (has-neighbor? x dir)
  (let ((pos (offset (get-position x) (get-offset dir)))
        (polyomino (get-polyomino x)))
    (and (in-range? pos (get-dimensions polyomino))
         (get-tile polyomino pos))))

Law of Demeter violation!
Branch: (method-call (draw-border (dc? dc) (tile? tile)))
Previous: (has-neighbor? #<object:tile?> north)
Message: (in-range? #<object:position?> #<object:dimensions?>)
Illegal supplier: #<object:dimensions?>

```

Figure 18: A Law of Demeter violation in the domino puzzle program.

```

(define-predicate (has-neighbor? x dir)
  (let ((pos (offset (get-position x) (get-offset dir)))
        (polyomino (get-polyomino x)))
    (has-tile? polyomino pos)))

(define-predicate (has-tile? polyomino pos)
  (and (in-range? pos (get-dimensions polyomino))
       (get-tile polyomino pos)))

```

Figure 19: A refactoring that obeys the Law of Demeter.

selected-polyomino-off-board? predicate method, not the after-method that uses the predicate.

6.4 Comparison with AspectJ implementation

Lieberherr, Lorenz, and Wu [37] present an AspectJ implementation of a run-time checker for the Law of Demeter over Join Points. While many of the details are different (due to the complexity of the AspectJ join point model), the basic outline is similar: record the set of potential preferred suppliers for each join point, and check that the target of the join point is contained in that set. There are two differences worth discussing, however, due to the different formulations of the Law of Demeter (and perhaps also due to the differences in expressiveness of AspectJ and Socrates).

The first difference is in the handling of newly-created objects. The Law of Demeter for Socrates regards as potential preferred suppliers all objects created during the processing of the decision

```

(define-after-method (rotate!
                    (=> ((viewed selected-polyomino-off-board?) x)
                        (:= tile selected-tile))
                    (integer? n))
  (highlight tile))

Law of Demeter violation!
Branch: around: (method-call
  (rotate!
    (=> ((viewed selected-polyomino-off-board?) x) (:= tile selected-tile))
    (integer? n)))
Previous: (rotate! #<object:polyomino?> -1)
Message: (highlight #<object:tile?>)
Illegal supplier: #<object:tile?>

```

Figure 20: A false positive violation report.

point, i.e., the result of `alloc` anywhere in the control flow of the previous decision point. The Law of Demeter over Join Points, on the other hand, only allows objects created by sibling join points, i.e., the results of constructor calls in the same method body. Their implementation records the constructed objects using a `perflow` aspect that associates one aspect instance with each execution join point; the checker retrieves the potential preferred supplier set via `aspectOf()`, which returns the aspect instance for the most recent join point only. The Socrates implementation uses a field associated with each decision point to record the created objects, but it uses `dp-all-previous` to add the object to every decision point on the stack, not just the most recent one. In order to do something similar in AspectJ, an explicit stack would have to be maintained to hold all the aspect instances for the current control flow.

The second difference is in the handling of field values. Their implementation records the current value of each field of an object by advising all `set` join points and updating a hash table; these values are always considered potential preferred suppliers, regardless of whether the current method used the field to retrieve the value. On the other hand, if a method retrieves a value from a field, storing it in a local variable, but that field value changes, the old value is no longer considered a potential preferred supplier. The Socrates implementation only records message return values as associated values, whether they came from fields or not, and does not track field updates; there is no special kind of decision point for field access, so these could not even be detected (apart from the naming convention).

Part III

The Socrates implementation

The following chapters detail some parts of the Socrates implementation and their programming interface. Chapter 7 presents reflective procedures and messages, and describes the process by which the source expressions are expanded into primitive syntax. Chapter 8 explains the predicate implication analysis algorithm used by the Socrates implementation, which uses a partial evaluator to interleave the steps of the canonicalization-based algorithm described in chapter 3.

Chapter 7

Procedure and message reflection

7.1 Procedure reflection

Analyzing implication between predicate procedures involves inspecting the source expression and environment of each procedure. MzScheme does not provide reflective access to this information, so the **proc-src.ss** module provides the forms **lambda/src** and **case-lambda/src** to record the source expression and environment of the created procedure. Their syntax is identical to the syntax for **lambda** and **case-lambda**.

The procedure reflection interface is as follows:

- **(proc/src? *v*)** returns **#t** if *v* is a reflective procedure, **#f** otherwise.
- **(proc-expr *proc*)** returns the expression (a syntax object) associated with procedure *proc*.
- **(set-proc-expr! *proc stx*)** associates the expression *stx* (a syntax object) with procedure *proc*. This does not change the value of the procedure; it is useful for recording syntactic sugar (such as **method-call**) that expands into a **lambda/src** expression (which happens before **lambda/src** gets a chance to store it).

- (**store-proc-expr** *expr*) evaluates *expr* and returns its return value, associating the unexpanded expression with it using **set-proc-expr!**. **make-branch** and **make-around** use this to record the predicate expression.
- (**proc-prim** *proc*) returns the functionalized primitive-syntax expression (a syntax object) associated with procedure *proc*. See below for details.
- (**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*.

The source expression for a reflective procedure is stored in two forms: unexpanded, with all syntactic sugar intact (for **proc-expr**) and fully macro-expanded with statements removed (for **proc-prim**, which is what the predicate implication analysis algorithm uses). The former is converted to the latter in two stages.

First, the expression is expanded into primitive MzScheme syntax using the MzScheme procedure **local-expand**, which uses all macro definitions in the lexical context of the expression. This primitive syntax is as follows:

expr is one of

variable

(**lambda** *formals* *expr* ...¹)

(**case-lambda** (*formals* *expr* ...¹) ...)

(**if** *expr* *expr*)

(**if** *expr* *expr* *expr*)

(**begin** *expr* ...¹)

(**begin0** *expr* *expr* ...)

(**let-values** (((*variable* ...) *expr*) ...) *expr* ...¹)

(**letrec-values** (((*variable* ...) *expr*) ...) *expr* ...¹)

(**letrec-syntaxes+values** *syntaxes* (((*var* ...) *expr*) ...) *expr* ...¹)

(**set!** *variable* *expr*)

(**quote** *datum*)

(**quote-syntax** *datum*)

```
(with-continuation-mark expr expr expr)
(#%app expr ...1)
(#%datum . datum)
```

case-lambda is a generalization of **lambda** for procedures with multiple arities. **begin0** is like **begin** but its values are the values of the first expression instead of the last. **let-values** and **letrec-values** are generalizations of **let** and **letrec** for expressions that can return multiple values; **letrec-syntaxes+values** also allows local syntax bindings. **quote-syntax** is like **quote** but produces a syntax object. **with-continuation-mark** is used to annotate the current continuation, e.g. to add source information for debuggers. **#%app** is inserted by MzScheme's macro expander to represent non-macro applications. **#%datum** is inserted by MzScheme's macro expander for literal constants (e.g. numbers and strings). An additional primitive syntactic form, **#%top**, is not shown; it is usually inserted by MzScheme's macro expander to represent top-level variables, but the call to **local-expand** uses **#%top** as a stop-identifier so that it will not be inserted in this case. This allows all variables to be treated the same, whether they are top-level, module-defined, or lexically scoped.

Second, all statements (expressions that are evaluated for side-effect only) are removed, because the predicate implication analysis makes the simplifying assumption that no side effects affect a predicate's return value:

- (**lambda** *formals stmt ... expr*) is replaced with (**lambda** (*formals expr*)).
- (**case-lambda** (*formals stmt ... expr*) ...) is replaced with (**case-lambda** (*formals expr*) ...).
- (**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*) ...) *expr1*). Note that binding forms with zero variables are removed.
- (**letrec-values** (((*var ...*) *expr*) ...) *stmt ... expr1*) is replaced with (**letrec-values** (((*var ...¹*) *expr*) ...) *expr1*).

- **(letrec-syntaxes+values *syntaxes* (((*var* ...) *expr*) ...) *stmt* ... *expr1*)** is replaced with **(letrec-syntaxes+values *syntaxes* (((*var* ...¹) *expr*) ...) *expr1*)**.
- **(set! *var* *expr*)** is replaced with **(#%app void)**. (This returns MzScheme's unique **void** value used for the value of expressions that have unspecified results in *R⁵RS* [34].)
- **(with-continuation-mark *key* *mark* *expr*)** is replaced with *expr*.

In addition, to regularize the syntax a bit more,

- **(lambda *formals* *expr*)** is replaced with **(case-lambda (*formals* *expr*))**.
- **(if *test* *then*)** is replaced with **(if *test* *then* (%app void))**.
- **(letrec-syntaxes+values *syntaxes* (((*var* ...¹) *expr*) ...) *expr1*)** is replaced with **(letrec-values (((*var* ...¹) *expr*) ...) *expr1*)**.

Thus, the syntax for expressions returned by **proc-prim** is as follows:

expr is one of

```

variable
(case-lambda (formals expr) ...)
(if expr expr expr)
(let-values (((variable ...1) expr) ...) expr)
(letrec-values (((variable ...1) expr) ...) expr)
(quote datum)
(quote-syntax datum)
(%app expr ...1)
(%datum . datum)

```

7.2 Message reflection

Messages are also procedures, and can be used as predicates; thus, the **msg.ss** module provides the procedure **make-msg/src** that returns a new message that allows reflective access

to its source expression and environment. (The **msg-sugar.ss** module provides the syntactic form **make-msg/src** to infer its name from context; the underlying procedure is renamed to **make-msg/src***.) In particular, **proc/src?** returns true for these messages, as does **msg/src?**. The **make-predicate-msg** form (section 4.10) also returns reflective messages.

Since the behavior for a message is spread out among multiple branches in the dispatch table, there is no single expression that can be used as the source. Instead, when applied to a message, **proc-prim** constructs an expression that's equivalent to the behavior of the message based on the current state of the dispatch table; essentially, the predicates and bodies of all the branches that could apply to the message are combined into one big **if** expression. Likewise, **proc-env-vars** and **proc-env-value** refer to a constructed environment for this expression. Note that the predicate branch bodies must themselves be reflective procedures for their source to be examined by the implication analysis algorithm; typically, they will be created with the **lambda-context** form (section 4.2), which uses **lambda/src**.

Chapter 8

Predicate implication analysis algorithm

The predicate implication analysis algorithm used in the Socrates implementation is a bit more complicated than the one described in section 3, in order to be more efficient. Essentially, it interleaves the various steps rather than traversing expressions multiple times, which might produce large intermediate expressions that will just be ignored later. The algorithm is structured as a **partial evaluator** for functional Scheme expressions.

The basic concept behind the algorithm is based on the definition of logical implication: predicate $p1$ implies a predicate $p2$ if, for all arguments, either $p1$ returns false or $p2$ returns true. This is equivalent to checking whether 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)))
```

Deciding predicate implication thus reduces to tautology checking.

A procedure is a tautology if the negation of its body expression will always evaluate to false regardless of the procedure's arguments. Socrates tries to determine this by partially evaluating the negated body, with the procedure's formal arguments considered dynamic and its free

variables considered static; if the result is the constant false, then the procedure is a tautology. Tautology checking thus reduces to partial evaluation.

The reflection interface defined in chapter 7 provides access to the current values of a procedure's free variables; these values are used during partial evaluation to reduce fully-static subexpressions to constants. In other words, Socrates uses an *online* partial evaluation strategy, similar to FUSE [53]. It is assumed that a procedure's free variables remain constant; most of these will refer to messages or Scheme procedures. Reducing application subexpressions containing these free variables can be thought of as a form of procedure inlining.

8.1 Input syntax

The input to the partial evaluator is a Scheme expression obeying the statement-free primitive syntax described in section 7.1. As an example of this primitive syntax, the implication procedure defined above will be converted 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)))))
```

8.2 Partial values

A partial evaluator for Scheme expressions is similar to a recursive Scheme interpreter, except that it produces **partial values** rather than Scheme values. A partial value is either a conditional or a **partial atom**; a partial atom is either a constant, an application, a closure, or an unknown. A conditional has a test partial atom and consequent and alternate partial values; an application has an operator partial atom and a list of argument partial atoms; a closure has a list of cases and a **partial environment**, which maps variables to partial values. A closure case has a formals form and a body expression.

The `pval.ss` module provides the following interface for inspecting partial values:

- `(pval? v)` returns `#t` if `v` is a partial value, `#f` otherwise.

- `(pif? v)` returns `#t` if v is a conditional, `#f` otherwise.
- `(pif-test pif)` returns the test partial atom of conditional pif .
- `(pif-then pif)` returns the consequent partial value of conditional pif .
- `(pif-else pif)` returns the alternate partial value of conditional pif .
- `(patom? v)` returns `#t` if v is a partial atom, `#f` otherwise.
- `(pconst? v)` returns `#t` if v is a constant, `#f` otherwise.
- `(pconst-value pconst)` returns the value of constant $pconst$.
- `(papp? v)` returns `#t` if v is an application, `#f` otherwise.
- `(papp-op papp)` returns the operator partial atom of application $papp$.
- `(papp-args papp)` returns the list of argument partial atoms of application $papp$.
- `(pclosure? v)` returns `#t` if v is a closure, `#f` otherwise.
- `(pclosure-pcases pclosure)` returns the list of cases of closure $pclosure$.
- `(pclosure-penv pclosure)` returns the partial environment of closure $pclosure$.
- `(pcase? v)` returns `#t` if v is a closure case, `#f` otherwise.
- `(pcase-formals pcase)` returns the formals form (a syntax object) of closure case $pcase$.
- `(pcase-body pcase)` returns the body expression (a syntax object) of closure case $pcase$.
- `(punknown? v)` returns `#t` if v is an unknown, `#f` otherwise.

The following utility procedures are also provided:

- `(ptrue? v)` returns `true` if v is a partial value that will always evaluate to true, `#f` otherwise.
- `(pfalse? v)` returns `#t` if v is a partial value that will always evaluate to false, `#f` otherwise.
- `(ppair? v)` returns `#t` if v is a pair constant or an application of `cons`, `#f` otherwise.

- `(pnull? v)` returns `#t` if v is a null constant, `#f` otherwise.
- `(papp-of? v prim)` returns `#t` if v is an application of a constant whose value is $prim$, `#f` otherwise.
- `(papp-tautology? v)` returns `#t` if v is an application of a constant primitive that is a tautology (see section 3.4), `#f` otherwise.
- `ptrue` is the constant `#t`.
- `pfalse` is the constant `#f`.
- `pnull` is the constant `null`.

8.3 Auxiliary data

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

- the current partial environment;
- a list of closure cases currently being applied, to avoid infinite loops;
- lists of partial atoms assumed true and false, for reducing conditional expressions.

At the top level (when partially evaluating an implication procedure), the partial environment maps the procedure's free variables to constant partial atoms holding their current values. The closure and assumption lists begin empty.

8.4 Partial evaluation

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 (an identifier) looks up the variable in the current partial environment to retrieve a partial value. Since this partial value resulted from a partial evaluation with different assumptions, it is re-checked against the current assumptions, which may reduce it further (see section 8.7).
3. Partially evaluating a **let-values** binding expression partially evaluates the bound expressions, binds them to the variables, then partially evaluates the body expression in this extended partial environment. For a binding clause that binds multiple values, each variable is bound to a **call-with-values** application that extracts the corresponding value from the partially evaluated bound expression.
4. Partially evaluating a **letrec-values** binding expression binds the variables to an undefined constant, partially evaluates the bound expressions in this extended partial environment, replaces the variable bindings with the resulting partial values, and then partially evaluates the body expression in the same partial environment. Multiple value bindings are handled as in the **let-values** case.
5. Partially evaluating a conditional expression (an **if** form) partially evaluates the test expression; if the result is a partial atom known to be true or false based on the current assumptions (see section 8.7), then the consequent or alternate expression, respectively, is partially evaluated, and the resulting partial value is returned. The consequent and alternate expressions are then both partially evaluated, but with the test partial atom added to the true or false assumptions, respectively. The resulting partial values are combined with the test partial atom into a conditional, which might be further simplified according to the following equalities:

- $(\text{if } A \ B \ B) = B$
- $(\text{if } A \ A \ \#f) = A$

If the test partial value is a conditional, then the consequent and alternate expressions are distributed across the conditional (see section 8.5).

6. Partially evaluating an abstraction expression (a **case-lambda** form) returns a closure encapsulating the abstraction's cases and the current partial environment.
7. Partially evaluating an application expression (an **#%app** form) causes the operator and argument expressions to be partially evaluated; if the resulting partial values are

all partial atoms, then the operator is **partially applied** to the list of argument values (see section 8.6). Otherwise, the partial application is distributed across conditionals (see section 8.5).

8.5 Distribution across conditionals

The test of a conditional partial value and the operator and arguments of an application partial atom must all be partial atoms; when partially evaluating conditional and application expressions, nested conditional expressions in these positions need to be pulled out according to the distribution rules in step 4 of section 3.1. The following procedure uses continuation-passing style to abstract the distribution operation:

```
(define (distribute-if pval k)
  (if (pif? pval)
      (reduce-if (pif-test pval)
                 (lambda () (distribute-if (pif-then pval) k))
                 (lambda () (distribute-if (pif-else pval) k)))
      (k pval)))
```

The `reduce-if` procedure takes a partial atom, a true thunk, and a false thunk; it either calls one of the two thunks, if the partial atom is known to be true or false, or calls both thunks with the partial atom added to the true or false assumptions and combines the results into a conditional.

When partially evaluating a conditional expression, `distribute-if` is called with the test partial value and a continuation that passes the result to `reduce-if` with thunks for partially evaluating the consequent and alternative expressions. When partially evaluating an application expression, `distribute-if` is mapped over the operator and argument list with a continuation that partially applies the first result to the rest, using the following version of `map` in continuation-passing style:

```
(define (map-k f l k)
  (if (null? l)
      (k null)
```

```
(f (car l) (lambda (v)
      (map-k f (cdr l) (lambda (l)
        (k (cons v l))))))))
```

8.6 Partial application

Partially applying an operator partial atom to an argument list of partial atoms is defined by the following rules:

1. If the operator is a 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 not currently being applied, the formal variables are bound to the corresponding argument partial atoms and the body expression of the case is partially evaluated; otherwise, the operator and arguments are combined into an application partial atom.
2. If the operator is a constant whose value is a reflective procedure (see section 7), the procedure's expression is partially evaluated to a closure, which is then partially applied to the argument list as in rule 1.
3. If the operator is a constant whose value is a primitive procedure, its partial procedure is determined (see section 8.8) and applied to the argument list. If this returns false, the operator and arguments are combined into an application partial atom.
4. Otherwise, the operator and arguments are combined into an application partial atom.

In each case, if an application partial atom is created and the application is known to be false based on the current assumptions (see section 8.7), then the constant false is returned instead.

A partial atom can be applied to a dotted argument list, i.e. one whose terminator is not the empty list. This can happen two ways: tautology checking partially applies a closure to an unknown to partially evaluate its body expression, and the partial procedure for `apply` partially applies its first argument to its middle arguments `consed` onto its last argument. (Note that a partial atom by itself is also a dotted list, with length zero, since it is neither null nor a pair.)

- When partially applying a closure case to a dotted argument list, additional arguments are constructed from the terminator to match the case's formals form using `car` and `cdr`. For example, partially applying a closure with a case whose formals form is $(a\ b\ c\ .\ d)$ to the dotted argument list $(x\ .\ y)$ binds a to x , b to the application $(\text{car } y)$, c to $(\text{car } (\text{cdr } y))$, and d to $(\text{cdr } (\text{cdr } y))$. If multiple closure cases might be applicable, applications are constructed to test the length of the terminator, and conditionals are constructed using these tests and the results of applying each case.
- Partially applying a primitive procedure with fixed arity to a dotted list uses the same method to construct the extra arguments, but if the procedure has multiple or variable arity, the partial procedure is not applied and the partial application returns an application of `apply`. For example, partially applying `vector` to the dotted argument list $(x\ .\ y)$ returns the application $(\text{apply } \text{vector } x\ y)$.

8.7 Checking the current assumptions

Several of the rules in section 8.4 require determining whether a partial atom is known to be true or false based on the current assumptions:

- A partial atom is known to be true if it is a true constant, a closure, an application of a constant primitive that always returns true, or implied by one of the true assumptions.
- A partial atom is known to be false if it is a false constant, implies one of the false assumptions, or is disjoint from one of the true assumptions.

The implication and disjointness relations between partial atoms mirrors those between atomic expressions as defined in section 3.2. Equivalence between partial values is similar to syntactic equivalence (see section 3.5): constants are equivalent if their values are `equal?`; conditionals and applications are equivalent if their components are equivalent; closures and unknowns are only equivalent to themselves. Substitution inside a partial atom re-applies applications of primitives in case they become further reducible due to the substitution; for example, substituting the constant 3 for the unknown x in the application $(\text{integer? } x)$ (because $(\text{eq? } x\ 3)$ is in the true

assumptions) applies the primitive procedure for `integer?` to 3 because it is a constant, which returns constant true.

When a variable's partial value is retrieved from a partial environment, it is re-checked against the current assumptions. If it is a conditional, its test partial atom is re-checked; if the resulting partial atom is known to be true or false, the consequent or alternate is re-checked, respectively, otherwise they are both re-checked and recombined with the re-checked test into a new conditional (possibly simplified using the equalities in rule 5 from section 8.4). If a partial atom being re-checked is known to be false, it is replaced by the constant false.

8.8 Partial procedures

Every primitive procedure has an associated **partial procedure** that takes some number of partial atoms and returns either a partial value or false. The `pproc.ss` module provides the following interface to this association:

- `(prim-pproc prim)` returns the partial procedure associated with *prim*.
- `(set-prim-pproc! prim pproc)` replaces the partial procedure associated with *prim* with *pproc*.
- `(default-partial-proc prim)` returns the default partial procedure for *prim*, which returns false if there are no arguments, or if not all of the arguments are constants; otherwise, it applies *prim* to the constant values and returns the result as a constant partial atom.
- `(define-partial (name . formals) body-expr ...1)` is equivalent to `(set-prim-pproc! name (lambda formals body-expr ...1))`, except that it can appear in either an expression context or a definition context.
- `(prim-apply prim patoms)` applies the partial procedure associated with *prim* to the list *patoms*; if it returns false, an application is created. If this application is known to be false, the constant false is returned instead.
- `(pcons patom1 patom2)` returns `(prim-apply cons (list patom1 patom2))`.

```

(define-partial (not patom)
  (reduce-if patom (lambda () pfalse) (lambda () ptrue)))

(define-partial (car patom)
  (if (papp-of? patom cons)
      (car (papp-args patom))
      ((default-partial-proc car) patom)))

(define-partial (cdr patom)
  (if (papp-of? patom cons)
      (cadr (papp-args patom))
      ((default-partial-proc cdr) patom)))

(define-partial (context-bindings patom)
  (if (papp-of? patom make-context*)
      (context-bindings (apply make-context* (papp-args patom)))
      ((default-partial-proc context-bindings) patom)))

(define-partial (eq? patom1 patom2)
  (if (eq? patom1 patom2)
      ptrue
      ((default-partial-proc eq?) patom1 patom2)))

(define-partial (assq patom palist)
  (cond ((pnull? palist)
         pfalse)
        ((ppair? palist)
         (let ((ppair (pcar palist)))
             (reduce-if (prim-apply eq? (list patom (pcar ppair)))
                        (lambda () ppair)
                        (lambda () (prim-apply assq (list patom (pcdr palist)))))))
        (else
         ((default-partial-proc assq) patom palist))))

```

Figure 21: Some partial procedures defined by Socrates.

- (pcar *patom*) returns (prim-apply car (list *patom*)).
- (pcdr *patom*) returns (prim-apply cdr (list *patom*)).

Figure 21 shows some partial procedures defined by Socrates.

8.9 Memoization

Many of the procedures in the partial evaluator implementation are memoized, i.e. their results are cached. For example, partial application atoms are reused when the same operator is partially applied to the same atoms. Besides the obvious efficiency benefit of avoiding repeated computation and duplicate storage, this also makes equality tests simpler: they can just use `eq?` rather than performing recursive traversals. This also the somewhat surprising effect that partial values are not just trees but can have shared structure, and can even contain cycles if an expression involves recursion. The `pval->datum` procedure (in the `pval-utils.ss` module) preserves this shared structure when converting a partial value to a Scheme datum representing the corresponding expression; this shared structure can be indicated by MzScheme's printer using the `print-graph` parameter:

```
> (define fact (lambda/src (x) (if (zero? x) 1 (* x (fact (sub1 x))))))
> (print-graph #t)
> (pretty-print (pval->datum (proc-pclosure fact)))
#1=(case-lambda
  ((x)
   (if (#%app (#%datum . #<primitive:zero?>) x)
       (#%datum . 1)
       (#%app
        (#%datum . #<primitive:*>)
        x
        (#%app #1# (#%app (#%datum . #<primitive:sub1>) x))))))
```

Here, the `#1#` in the partial application on the last line refers to the entire partial closure value (assigned to `#1` on the first line).

Part IV

Analysis

Chapter 9

Related work

9.1 Foundations

The research presented in this paper started with the observation of the similarities between the predicate dispatching language described by Ernst, Kaplan, and Chambers [20] and the dynamic join point model of AspectJ [35], particularly the notion of pointcuts as predicates. It also became clear 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 [46]. A prototype language called Fred [47] was developed that implemented this basic extension, which evolved into Socrates as defined in part I. This section summarizes the differences between Socrates and predicate dispatching and between Socrates and AspectJ.

9.1.1 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” [20]. Socrates provides analogues to all the primitives and syntactic sugar defined by their language, including methods (with formal patterns and predicates

guards), predicate abstractions (that can return name bindings), and classifiers.

Socrates goes a step further to also 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.

The authors point out some limitations to their predicate implication analysis 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 analysis algorithm may fail when comparing them due to the termination heuristics of the algorithm (see section 3).

Their implication analysis algorithm, like that of Socrates, is based on tautology checking. It proceeds in three stages:

1. Predicate abstractions are inlined and variable bindings are expanded. This reduces the predicate to a logical formula over atoms (class tests and black box expression tests).
2. Implication relations are computed among the atoms in the formula, based on the class hierarchy and syntactic equivalence of the tested expressions.
3. The formula is evaluated with the atoms assigned every combination of truth values that are consistent with the implication relations. If the formula evaluates to true for every consistent truth assignment, it is a tautology.

(Chambers and Chen [11] later present a refinement that avoids truth assignments by converting the formula to disjunctive normal form and using structural recursion.) The partial evaluation strategy of Socrates’s implication analysis algorithm is a generalization of their algorithm, able to deal with arbitrary expressions involving conditionals, applications, and abstractions. In particular, it can handle higher-order predicates. It is also potentially more efficient than the canonicalization algorithm, because the interleaving of stages allows it to avoid processing subexpressions in unreachable branches of conditionals. See 9.4 for connections to other partial evaluation research.

In Socrates, messages, predicates, and branches are first-class values that 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 a primitive construct in the core semantics of predicate dispatching; the main body of [20] 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).

Figures 22 through 26 compare the syntax of Socrates with the predicate dispatching language using examples from the predicate dispatching paper.

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.

There have been other implementations of predicate dispatching, but none have included the complete dynamically-typed subset of the language defined in [20]. Gd [33] is a Java implementation of an interpreter for a simple object-based language with generic functions and

```

type Expr;
-- default constant-fold optimization: do nothing
method ConstantFold(e) { return e; }

type AtomicExpr subtypes Expr;
  class VarRef subtypes AtomicExpr { ... };
  class IntConst subtypes AtomicExpr { value:int };
  ... -- other atomic expressions here

type Binop;
  class IntPlus subtypes Binop { ... };
  class IntMul subtypes Binop { ... };
  ... -- other binary operators here

class BinopExpr subtypes Expr { op:Binop, arg1:Expr, arg2:Expr, ... };
-- override default to constant-fold binops with constant arguments
method ConstantFold(e@BinopExpr{ op@IntPlus, arg1@IntConst, arg2@IntConst }) {
  return new IntConst{ value := e.arg1.value + e.arg2.value }; }
... -- more similarly expressed cases for other binary and unary operators here

(define-class expr? () ())
(define-method (constant-fold e) e)

(define-class atomic-expr? (expr?) ())
(define-class var-ref? (atomic-expr?) (...))
(define-class int-const? (atomic-expr?) (value))

(define-class binop? () ())
(define-class int-plus? (binop?) ())
(define-class int-mul? (binop?) ())

(define-class binop-expr? (expr?) (op arg1 arg2 ...))
(define-method (constant-fold (=> (binop-expr? e)
  (int-plus? op) (int-const? arg1) (int-const? arg2))))
  (make int-const? (+ (get-value (get-arg1 e)) (get-value (get-arg2 e)))))

```

Figure 22: Pattern-matching example from [20] translated into Socrates.

```

-- handle case of adding zero to a non-constant
method ConstantFold(e@BinopExpr{ op@IntPlus, arg1=a1, arg2=a2 })
  when (a1@IntConst{ value=v } and test(v == 0)
        and not(a2@IntConst) and let res := a2)
  or (a2@IntConst{ value=v } and test(v == 0)
        and not(a1@IntConst) and let res := a1) {
  ... -- increment counter, or do other common work here
  return res; }

(define-method (constant-fold (=> (binop-expr? e) (int-plus? op) (:= a1 arg1) (:= a2 arg2)))
  & (or:= (and:= (=> (int-const? a1) (:= v value)) (= v 0)
               (not (int-const? a2)) (:= res a2))
      (and:= (=> (int-const? a2) (:= v value)) (= v 0)
             (not (int-const? a1)) (:= res a1)))
  res)

```

Figure 23: Boolean expressions example from [20] translated into Socrates.

```

predicate on_x_axis(p@point)
  when (p@cartesianPoint and test(p.y == 0))
  or (p@polarPoint and (test(p.theta == 0) or test(p.theta == pi)));

method draw(p@point) { ... }      -- draw the point
method draw(p@on_x_axis) { ... }  -- use a contrasting color so point is visible

(define-predicate (on-x-axis? (point? p))
  (or (and (cartesian-point? p) (= (get-y p) 0))
      (and (polar-point? p) (or (= (get-theta p) 0) (= (get-theta p) pi)))))

(define-method (draw (point? p)) ...)
(define-method (draw (on-x-axis? p)) ...)

```

Figure 24: Predicate abstraction example from [20] translated into Socrates.

```

predicate LoopExit(n@CFG_2succ{ next_true: t, next_false: f })
  when (test(t.is_loop_exit) and let nl := t and let ne := f)
  or (test(f.is_loop_exit) and let nl := f and let ne := t)
  return { loop := nl.containing_loop, next_loop := nl, next_exit := ne };

(define-predicate (loop-exit? (=> (cfg-2succ? n) (:= t next-true) (:= f next-false)))
  & (or:= (and:= (get-is-loop-exit t) (:= nl t) (:= ne f))
      (and:= (get-is-loop-exit f) (:= nl f) (:= ne t)))
  (return (loop (get-containing-loop nl)) (next-loop nl) (next-exit ne)))

```

Figure 25: Name-binding example from [20] translated into Socrates.

```

class Window { ... }

classify(w@Window)
  as Iconified when test(w.iconified)
  as FullScreen when test(w.area() == RootWindow.area())
  as Big when test(w.area() > RootWindow.area()/2)
  as Small otherwise;

method move(w@FullScreen, x@int, y@int) { }      -- nothing to do
method move(w@Big, x@int, y@int) { ... }        -- move a wireframe outline
method move(w@Small, x@int, y@int) { ... }      -- move an opaque window
method move(w@Iconified, x@int, y@int) { ... } -- modify icon coordinates

-- resize, maximize, and iconify similarly test these predicates

(define-class window? () ( ... ))
(classify ((window? w))
  (iconified? (get-iconified? w))
  (full-screen? (= (get-area w) (get-area root-window)))
  (big? (> (get-area w) (/ (get-area root-window) 2)))
  (small? otherwise))

(define-method (move (full-screen? w) (integer? x) (integer? y)) )
(define-method (move (big? w) (integer? x) (integer? y)) ... )
(define-method (move (small? w) (integer? x) (integer? y)) ... )
(define-method (move (iconified? w) (integer? x) (integer? y)) ... )

```

Figure 26: Classifier example from [20] translated into Socrates.

predicate-guarded methods. Ucko [63] used the meta-object protocol of CLOS to add predicates as method qualifiers. JPred [42] is an extension to the Java language that uses a theorem prover to analyze predicate implication. The former two do not include field pattern matching, the latter two do not include predicate abstractions, and none of the three include classifiers. Also, JPred does not allow arbitrary expressions in predicates, although its predicate language does include binary operators for checking equality and inequality of values as well as for performing rational linear arithmetic.

9.1.2 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. Socrates only has one kind of decision point: message send, which is equivalent to AspectJ's method call. Most of the other join points are not needed in Socrates, since object construction, initialization, and field reference are all done by sending messages. A branch body execution decision point could be added to Socrates analogous to AspectJ's method execution join point, but it is unclear how useful this would be, and whether the benefit would outweigh the loss of the simplicity of having only one kind of join point.

Most of the primitive pointcut designators in AspectJ can be emulated by predicates in Socrates, and syntactic sugar for many of them is defined in section 4.5. AspectJ can also emulate most Socrates predicates by using the `if` pointcut designator, which can contain an arbitrary boolean Java expression. This expression can refer to the special `thisJoinPoint` variable, which is a reification of the current join point being considered by the pointcut, i.e. an object containing the target object and arguments to the method call, the current `this` object, and the type signature of the method. However, it does not contain any representation of the control stack, which is only available through the `cflow` and `cflowtop` pointcut designators. Socrates instead allows arbitrary inspection of the control stack through the `dp-previous` decision point accessor.

The `within` pointcut designator is somewhat problematic to emulate in Socrates; it refers to join points caused by expressions in the bodies of methods in a particular class declaration. The `dp-within` accessor in Socrates returns the branch whose body contains the message send

expression that caused the decision point, but branches are not contained in class declarations. Instead, a predicate could check that the branch was defined by a particular module or unit, but since MzScheme does not provide reflection facilities for module and unit definitions, these associations would have to be created manually (perhaps with the help of syntactic sugar).

The `withincode` pointcut is somewhat more straightforward; it refers to join points caused by expressions defined in the declaration of a method or constructor that matches a signature pattern. Since Socrates has no static typing, the analogue of signatures is predicate expressions, which could be compared with *pred-equivalent?* defined as follows:

```
(define (pred-equivalent? p1 p2)
  (and (pred-implies? p1 p2) (pred-implies? p2 p1)))
```

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 execution 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 pointcut expression 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. Likewise, there is no special Socrates construct corresponding to aspect instances, but these can be simulated with plain objects and an *aspect-of* field. For per-object aspects (i.e., those defined with `perthis` or `pertarget`), this field would associate aspect instances with objects; for per-control-flow aspects (i.e., those defined with `percflow` or `percflowbelow`), this field would associate aspect instances with decision points. The Law of Demeter checker implementation described in section 6.2 uses both of these approaches, but with two fields that directly hold the information, rather than encapsulating them into aspect instances.

The Socrates model demonstrates that aspect-oriented programming can be viewed as a generalization of object-oriented programming, and that branches can emulate both methods and advice. In fact, branches are almost equivalent to advice, which means that advice is almost a generalization of methods. One could in fact write programs in AspectJ with no methods at all, where instead all of the behavior is supplied by advice. For example, a factorial method could be implemented in AspectJ as in figure 27. Note that it still must include a stub method declaration for `fact`, so that there is a static method signature for the compiler to check for calls to `fact`, but it can be regarded as equivalent to `define-msg`. Also note that the base case advice must be defined first, since intra-aspect advice precedence is determined solely by lexical ordering.

Wand, Kiczales, and Dutchyn [65] demonstrate a similar style of implementing recursive procedures using advice, although they separate concerns along a different axis: rather than separating the base case from the recursive case, they separate the recursion concern from the computation concern. In terms of the above example, their approach would have the `fact` method call `fact(n)` instead of being a stub, and the `factCall` advice would call `proceed(n-1)`. (Alternately, the `fact` method could remain a stub, and a third piece of advice could do the trivial recursion.)

```

aspect Fact {
    static int fact(int n) {
        throw new RuntimeException("Message not understood: fact(" + n + ")");
    }

    pointcut factBaseCase(int n):
        factCall(n) &&
        if(n == 0);
    int around(): factBaseCase(int) { return 1; }

    pointcut factCall(int n):
        execution(int fact(int)) &&
        args(n);
    int around(int n): factCall(n) { return n * fact(n-1); }
}

```

Figure 27: A Socrates-style implementation of factorial in AspectJ.

9.2 Other AOP models

9.2.1 Hyperspaces

The **hyperspaces** model [49] is an approach to achieving **multi-dimensional separation of concerns** [59]: “flexible and incremental separation, modularization, and integration of software artifacts based on any number of concerns”. Rather than submitting to the “tyranny of the dominant decomposition”, in which all code must be organized into a single hierarchy, a program can have many different decompositions along different dimensions of concern. A hyperspace is a multi-dimensional matrix of program units (such as classes and methods), where each dimension is partitioned into concerns; each unit belongs to exactly one concern in each dimension. The units for a concern can be encapsulated into a set called a **hyperslice**. A hyperslice can be any set of program units; in particular, it can contain methods from different classes, and methods from a single class can be spread across multiple hyperslices. Hyperslices can be collected into **hypermodules**, which contain a set of **integration relationships** that specify how the hyperslices should be composed; for example, two methods with the same name from different hyperslices can be combined into single methods with the behavior of both, or one method can be specified to override the other. Hyper/J [58] is a tool supporting hyperspaces, providing a language for defining concern dimensions, mapping concerns to Java classes and

methods, and specifying hyperslices and hypermodules in terms of the concern dimensions.

While Socrates does not provide syntactic sugar to emulate the language features of Hyper/J directly, it shares the philosophy of flexible separation of concerns: program units can be organized by concern, rather than only by class. Also, both Socrates and the hyperspaces model are symmetric [31], in that all program units are considered equal, as opposed to an asymmetric model such as AspectJ, in which methods and advice are two different constructs. (Hyper/J still has two different kinds of method composition, though—class inheritance and integration relationships—while Socrates just has **follow-next-branch**.) A hyperslice can be emulated by a MzScheme module or unit [24] containing the messages, predicates, and branches for a particular concern, referencing other concerns through its imports. Compound modules and units in MzScheme provide some of the functionality of hypermodules, including simple integration relationships such as renaming.

Other integration relationships that can be specified in a hypermodule are similar in spirit to the branch precedence mechanisms in Socrates. The difference is that integration relationships are specified separately from the program unit declarations that they refer to, whereas behavior composition in Socrates is determined by properties of the branches themselves, such as predicate implication, around-branches, and the use of **follow-next-branch** in the body (explicitly, or implicitly in the syntactic sugar for **make-before** and **make-after**). For example, the **merge** and **bracket** relationships in Hyper/J specify that multiple methods are to be executed sequentially, while the **override** relationship specifies that one method overrides some others. This style of explicit composition can be more cumbersome than relying on implicit composition based on branch properties, but can also be more flexible: for example, the **order** relationship in Hyper/J can be used to customize precedence between program units, and the **summary function** relationship can be used to synthesize return values from a set of merged methods, whereas in Socrates a branch can only process the return value from the next-most-precedent branch.

Hyper/J has limited forms of dynamic reflection: a **bracket** relationship declaration can use the **\$OperationName** and **\$ClassName** keywords to pass the name of the bracketed operation and class to the bracket methods. It can also include a callsite specification that limits the bracketing to certain dynamic calling contexts; this is enough to help address the “jumping aspects” problem [9], but is not as flexible as being able to inspect the full control flow stack

with `dp-previous` in Socrates.

9.2.2 Composition filters

The **composition filters** model [3, 6] is an aspect-oriented technique where aspects are expressed as filters attached to a class (or a set of classes [5]) 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 28 shows some example filter specifications from [3] translated into Socrates. Figure 29 shows a test program and its output.

9.2.3 Classpects

The goal of the **classpects** module construct [52] is to unify classes and aspects. The Eos-U language extends C# [19] by adding a pointcut-like method binding construct to classes rather than adding separate aspect and advice constructs. This is somewhat similar to branches in Socrates, although there is still a separation between methods and bindings, and method signatures and pointcuts are separate concepts. Also, the binding construct language is not fully expressive: it has no equivalent to AspectJ's `if` pointcut designator. In addition, precedence between methods is limited to the usual object-oriented overriding relationship, i.e. subtyping of the method receiver, with `around` bindings having higher precedence.

9.3 Aspect calculi

One of the motivations behind Socrates was to define a simple core language that was general enough to model object-oriented and aspect-oriented paradigms. Several aspect calculi have been developed with a similar goal. Although Socrates does not have a formal semantics, it is worthwhile to compare its mechanisms to those of the aspect calculi.

```

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}};

(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 28: Example filter specifications from [3] translated into Socrates. (The line marked with an asterisk was changed from `recView => reply`, which is likely a mistake—a receiver should also be able to get the content of the mail.)

```

(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 29: A program to test the filter definitions in figure 28.

9.3.1 μ ABC

Bruns, Jagadeesan, Jeffrey, and Riely [8] present a small-step operational semantics for an aspect calculus called μ ABC. They demonstrate its expressiveness by presenting encodings of various other languages into μ ABC: the lambda calculus, MinAML (see section 9.3.2), a class-based language with advice, and a π -calculus extended with pointcuts.

μ ABC is similar to Socrates in that, rather than having separate constructs for base code and aspect code, there is only advice. Sending a message causes the list of current advice declarations to be consulted, and the most recently declared advice that is applicable is invoked. The parameters of the advice are the sender and receiver of the message and the list of other applicable advice (so that the advice body can proceed to the next advice). Messages and objects are both implemented by a partially-ordered set of names called **roles**. Advice can dynamically declare new roles or new advice.

The pointcut language is a boolean algebra over the sender and receiver of a message (as well as the message itself), with existential and universal quantification over roles (bounded by the role partial order). Essentially, pointcuts can check types, but they cannot actually send messages themselves; thus, none of the dynamic state of the abstract machine can be tested.

9.3.2 MinAML

Walker, Zdancewic, and Ligatti [64] present a small-step operational semantics for an aspect calculus that extends the simply-typed lambda calculus with explicitly labeled program points and first-class advice. They also present a simple aspect-oriented language called MinAML and a translation from it to the core aspect calculus. This translation converts function declarations into lambda expressions whose bodies are wrapped with entry and exit join points, which can be referred to by the pointcuts of advice declarations. They also extend the core aspect calculus with objects (using the object calculus of Abadi and Cardelli [1]) and more complex pointcuts, including label sets and stack patterns.

The core aspect semantics are parameterized by a relation determining whether a pointcut is satisfied by a configuration, i.e. the current state of the abstract machine. This could be any arbitrary predicate, like in Socrates, but MinAML only provides stack patterns, which are essentially regular expressions for matching against the current value of the stack. By default the stack includes the labels of the expressions being evaluated, but arbitrary values can also be pushed onto the stack. Stack patterns are used to emulate various pointcut designators from AspectJ, including `within`, `cflow`, and `cflowtop`.

Their first-class advice is similar to branches in Socrates: there is a global list of advice, and whenever a labeled expression is evaluated, the sublist of advice applicable to the current stack is determined and composed into a function that is applied to the expression value. Advice can either be prepended or appended to the current list, whose order determines the order of composition of applicable advice. (Advice cannot be removed from the list, although this is mentioned as future work.) A `return...to` expression essentially treats labels as continuations, which is used to implement advice that does not `proceed`, i.e., return to the next advice in the composition. (Advice in MinAML must have either zero or one `proceed` expression, but this restriction is only to simplify the presentation.)

9.3.3 PAC

Clifton, Leavens, and Wand [15] present a big-step operational semantics for a parameterized aspect calculus (PAC) that extends the object calculus of Abadi and Cardelli [1] with join points

and advice, parameterized by a pointcut description language. A program consists of a term and a (static) sequence of advice. Each of the three term reduction rules in the object calculus (object creation, method invocation, and method update) is replaced by a pair of reduction rules, one for when no advice matches the join point (which is otherwise identical to the object calculus reduction rule) and one for when some advice matches. A join point is a 4-tuple consisting of the join point kind (creation, invocation, or update), the evaluation context of the expression being reduced, the signature (list of method labels) of the target object of the expression, and the method label that is being invoked or updated (or ϵ for object creation). A piece of advice is a pair consisting of a pointcut description and a “naked method”, which has the form of a method but is not part of an object. Advice precedence is determined by the sequence order.

The pointcut description language semantics determines whether a pointcut description matches a join point; a sample pointcut description language is provided to emulate features of AspectJ. They also provide an encoding for AspectJ-style open classes (inter-type declarations), such as adding a field to a class by adding advice on object creation to add the field. They then provide a similar encoding for HyperJ-style composition of multiple objects into a single delegate object that forwards method invocations and method updates. Finally, they sketch an extension to the pointcut description language to model traversals in the style of DJ [48] by generating advice that traverses the fields of the target object.

While they do not provide a pointcut description language that can do arbitrary computation like predicates in Socrates, it would be straightforward to plug such a language into their semantics, perhaps even using PAC itself. In order to have access to the dynamic state, however, the join point and its evaluation context should include the target object of each invocation, rather than just its signature. This is equivalent to the `dp-args` field of a decision point (the object calculus emulates method arguments by encapsulating them as fields on the target object).

9.3.4 Extended CEKS machine

Tucker and Krishnamurthi [61] present a small-step semantics for a higher-order functional language extended with pointcuts and advice, based on the CEKS machine [21]. While it does not include object-oriented features, it is worth comparing to Socrates because it is based on a MzScheme implementation of advice with arbitrary predicate procedures as pointcuts [60]. It

extends the usual CEKS reduction rule for function application to check the **aspect environment**. A join point is simply the function being applied, and the argument to each pointcut predicate is a list of join points representing the current function call stack. This enables emulation of AspectJ's `call`, `cflow`, and `withincode`, but other pointcut designators are not emulatable because the function arguments are not available. Advice precedence is determined by its ordering in the aspect environment. Advice body can proceed to the next matching advice, if any, or else the original function body.

Advice can be installed into the aspect environment with lexical or dynamic scoping: the **around** form installs advice during the static extent of an expression, while the **fluid-around** form installs advice during the dynamic extent of an expression. Similar techniques could be used in Socrates to add branches to the global dispatch table during the static or dynamic extent of an expression and remove them afterward. The benefit of scoped advice is somewhat unclear to me, though, without a compelling example of its usefulness.

9.4 Partial evaluation

The partial evaluation algorithm used in Socrates is heavily influenced by Ruf's description of FUSE [53], a set of online partial evaluators for the functional subset of Scheme. The algorithm in Socrates has much simpler (and thus more restrictive) termination heuristics, however; it also leaves out many details relating to code generation and program specialization, which is a significant portion of the FUSE research but is irrelevant for the purposes of tautology checking.

On the other hand, none of the FUSE partial evaluators carry predicate assumptions while traversing conditional expressions, although this technique is mentioned in an earlier FUSE paper [66] as a hypothetical example of information usage. In fact, this is typically not considered to be a part of partial evaluation at all; it fits somewhere in between supercompilation and generalized partial computation, which are both considered to be generalizations of partial evaluation.

Turchin's **supercompilation** [62, 56] (short for **supervised compilation**) maintains a history of computation states (called **configurations**) during symbolic evaluation of a program in order

to generate a more efficient program. Secher [55] defines **positive supercompilation**, which propagates information down the positive branches of a conditional, and **perfect supercompilation**, which propagates information down both branches of a conditional. However, only equations ($x = y$) and disequations ($x \neq y$) are propagated, which is less general than Socrates, which propagates atomic predicate expressions down both branches of a conditional.

Futamura's **generalized partial computation (GPC)** [27, 26] combines partial evaluation with theorem proving. An environment of assertions is carried down both branches of a conditional, and a theorem prover is used to determine if a conditional can be reduced based on the assertions in the current environment. The theorem prover uses domain knowledge about the types of primitive functions, including arithmetic operations and inequalities. This is a much more general approach than the ad-hoc rules described in section 3.2. The distribution step of GPC [26, section 2.2] is essentially the same as the **distribute-if** procedure defined in section 8.5.

Chapter 10

Conclusion and future work

Aspect-oriented programming is usually presented as an extension of object-oriented programming, a set of mechanisms that can be added to support the separation of concerns that cut across the separation provided by the object-oriented mechanisms. Socrates demonstrates that the mechanisms that support separation of concerns in both aspect-oriented and object-oriented languages can be unified and generalized: methods, classes, advice, and inter-type declarations can all be emulated with messages and branches. Socrates is also a full implementation of dynamic predicate dispatching, and its predicate implication analysis is potentially more efficient than the canonicalization algorithms described in the predicate dispatching literature because it can avoid processing unreachable subexpressions.

The research presented in this dissertation can be extended in many different directions, some of which are outlined in the remainder of this chapter.

10.1 Standalone language

Socrates is embedded into MzScheme, which has many design and implementation benefits. However, it might also be useful to design a self-contained standalone language with branches and messages as the only primitive constructs. There are two possible approaches: make a

MzScheme module that only exports the language's primitive syntax, or make a full-blown interpreter or compiler.

The **socrates** language module re-exports all of MzScheme's standard variable bindings and syntax definitions, but these are not strictly needed. For example, **lambda** could be emulated with a message and a branch:

```
(define-syntax lambda
  (syntax-rules ()
    ((lambda formals . body)
     (let ((anonymous (make-msg)))
       (define-method (anonymous . formals) . body)
       anonymous))))
```

This would essentially turn every procedure call into a decision point. Potentially this could add flexibility by allowing branches to affect any procedure call, but it might also make it harder to write predicates that do not cause infinite loops.

An interpreter or compiler for a standalone language would have additional flexibility. The syntax of the language would not have to conform to the S-expression metasyntax; in particular, the pattern-matching syntax might be easier to use with an infix notation. An interpreter or compiler implementation would also not be tied to Scheme's model of environments and scoping.

10.2 Formal semantics

A formal semantics of the core Socrates language, either as an extension to the lambda calculus or as a calculus of just messages and branches (similar to the standalone language discussed above), would help clarify its relation to other languages and to aspect calculi. It would also be useful to be able to prove that the predicate implication analysis algorithm terminates, and that it is a conservative approximation.

10.3 Branch precedence

It is useful to have branch precedence be partly determined by predicate implication analysis, as a generalization of object-oriented method inheritance. However, the fact that around-branches are sometimes necessary to override this precedence relation points out that there might be a need for a more sophisticated mechanism for specifying branch precedence. This is related to aspect composition and conflict resolution, which is an active area of research. The hyperspaces model (see section 9.2.1) is one example, where composition rules can be defined to specify how units of behavior should be merged or overridden, separately from the behavior definition.

The implication analysis algorithm itself could also be strengthened to cover more cases where it currently cannot determine implication. One example is transitive operators: $(< x y)$ should imply $(< x z)$ if $(< y z)$. Another is that $(\text{cflow } p1)$ should imply $(\text{cflow } p2)$ if $p1$ implies $p2$. One approach would be to use a theorem prover to analyze implication and disjointness relationships between atomic expressions, similar to JPred [42] or generalized partial computation [27]. The latter also uses the theorem prover to support more sophisticated termination heuristics, which are also found in other partial evaluators such as FUSE [66].

10.4 Efficient implementation

The Socrates implementation is rather naive in places, and there is much room for improvements in efficiency. A large source of slowdown is repeated computation when evaluating predicates to determine which branches are applicable to a decision point. Chambers and Chen [11] discuss an algorithm to build an efficient dispatch tree for methods with predicate dispatching, using common subexpression elimination and other techniques. The partial evaluator could also be used to speed up predicate evaluation, although its simplifying assumption that mutation does not affect predicate return values may not be appropriate, since side effects during predicate evaluation should still happen even if they do not affect the return value. Another issue to consider is dynamic addition and removal of branches; this can be handled efficiently with intelligent caching, but it would also be interesting to investigate the design trade-offs of disallowing dynamic modifications to the dispatch table.

Implementing predicate dispatching's static type system would also help guide an implementation to remove many dynamic type tests. Alternately, type inference could be used.

10.5 Tool support

Socrates's support for separation of concerns allows the flexible organization of code according to concern. However, a particular branch is often associated with several concerns at once. Rather than a program being a set of text source files that divide the branches into one particular decomposition of concerns, the branch dispatch table could be made persistent across sessions and viewed as a source repository. Socrates provides a rudimentary reflective interface onto the table, but higher-level tools could be developed to allow the programmer to browse subsets of branches according to concern. Program development would involve an incremental cycle of adding branches to the table and modifying them.

Bibliography

- [1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.
- [2] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1996.
- [3] 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.
- [4] AOSD.net. Aspect-Oriented Software Development Community & Conference. Web site. <http://aosd.net/>.
- [5] 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.
- [6] L. M. Bergmans. *Composing Concurrent Objects*. PhD thesis, TRESE, University of Twente, Enschede, The Netherlands, 1994.
- [7] 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.

- [8] G. Bruns, R. Jagadeesan, A. Jeffrey, and J. Riely. muABC: A minimal aspect calculus. In P. Gardner and N. Yoshida, editors, *CONCUR 2004 – Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings*, volume 3170 of *Lecture Notes in Computer Science*, pages 209–224. Springer, 2004.
- [9] L. Carver. Using brackets to corral jumping aspects. In *Proceedings of the Workshop on Advanced Separation of Concerns, Conference on Object-Oriented Programming, Systems, and Languages (OOPSLA)*, 2000.
- [10] C. Chambers. Predicate classes. *Lecture Notes in Computer Science*, 707:268–??, 1993.
- [11] C. Chambers and W. Chen. Efficient multiple and predicate dispatching. In *Proceedings of the 1999 ACM Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA '99)*, volume 34(10) of *ACM SIGPLAN Notices*, pages 238–255, Denver, CO, November 1999. ACM.
- [12] J. Clements, P. T. Graunke, S. Krishnamurthi, and M. Felleisen. Little languages and their programming environments. In *Proceedings of Monterey Workshop 2001. Engineering Automation for Software Intensive System Integration*, Monterey, CA, June 2001. Naval Postgraduate School.
- [13] C. Clifton. Generalized open classes, multimethods, and modularity: Compilation, type-checking, and reasoning in MultiJava. Dissertation proposal, Iowa State University Department of Computer Science, Jan. 2003.
<http://www.cs.iastate.edu/~cclifton/research.shtml>.
- [14] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota*, volume 35(10), pages 130–145, 2000.
- [15] C. Clifton, G. T. Leavens, and M. Wand. Parameterized Aspect Calculus: A core calculus for the direct study of aspect-oriented languages. Technical Report 03-13, Department of Computer Science, Iowa State University, November 2003.
- [16] E. W. Dijkstra. On the role of scientific thought. Published as [18], Aug. 1974.
- [17] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

- [18] E. W. Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer-Verlag, 1982.
- [19] ECMA. Standard ECMA-334: C# language specification, 3rd edition, June 2005.
<http://www.ecma-international.org/publications/standards/Ecma-334.htm>.
- [20] 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.
- [21] M. Felleisen, D. P. Friedman, E. Kohlbecker, and B. Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52(3):205–237, 1987.
- [22] R. E. Filman. A bibliography of aspect-oriented programming, version 1.3. Technical Report 04.02, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffett Field, California, Sept. 2004.
- [23] M. Flatt. *PLT MzScheme: Language manual*, 299.100 edition, March 2005.
<http://www.plt-scheme.org/software/mzscheme/>.
- [24] M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 236–248, 1998.
- [25] M. Flatt, R. B. Findler, and J. Clements. *PLT MrEd: Graphical Toolbox Manual*, 299.100 edition, March 2005.
<http://www.plt-scheme.org/software/mred/>.
- [26] Y. Futamura, Z. Konishi, and R. Glück. WSDFU: Program transformation system based on generalized partial computation. In T. Mogensen, D. Schmidt, and I. H. Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation*, volume 2566 of *Lecture Notes in Computer Science*, pages 358–378. Springer-Verlag, 2002.
- [27] Y. Futamura, K. Nogi, and A. Takano. Essence of generalized partial computation. *Theoretical Computer Science*, 90(1):61–79, 1991.
- [28] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.

- [29] A. Goldberg and D. Robson. *Smalltalk 80 The Language*. Addison-Wesley, 1989.
- [30] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification*. Addison-Wesley, Reading, MA, USA, 1996.
- [31] W. Harrison, H. Ossher, and P. Tarr. Asymmetrically vs. symmetrically organized paradigms for software composition. Technical Report Research Report RC22685, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, December 2002.
- [32] J. G. Hosking, J. Hamer, and W. B. Mugridge. Integrating functional and object-oriented programming. In *Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS) 3*, pages 345–355, Sydney, 1990.
- [33] C. S. Kaplan. *Dubious with Predicate Dispatching (Güd) User's Manual*, Mar. 1998. <http://www.cs.washington.edu/research/projects/cecil/www/Gud/>.
- [34] R. Kelsey, W. Clinger, and J. Rees. Revised⁵ report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.
- [35] 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*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353, Budapest, Hungary, 2001. Springer-Verlag.
- [36] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.
- [37] K. Lieberherr, D. H. Lorenz, and P. Wu. A case for statically executable advice: Checking the Law of Demeter with AspectJ. In M. Aksit, editor, *Second International Conference on Aspect-Oriented Software Development*, Boston, 2003. ACM Press.
- [38] K. J. Lieberherr. Controlling the complexity of software designs. In J. Estublier and D. Rosenblum, editors, *International Conference on Software Engineering*, pages 2–11, Edinburgh, Scotland, 2004. ACM Press. invited presentation.
- [39] K. J. Lieberherr and I. Holland. Assuring good style for object-oriented programs. *IEEE Software*, pages 38–48, September 1989.

- [40] K. J. Lieberherr and I. Holland. Formulations and Benefits of the Law of Demeter. *SIGPLAN Notices*, 24(3):67–78, March 1989.
- [41] K. J. Lieberherr, I. Holland, and A. J. Riel. Object-oriented programming: An objective sense of style. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 323–334, San Diego, CA, September 1988.
- [42] T. Millstein. Practical predicate dispatch. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 345–364, New York, NY, USA, 2004. ACM Press.
- [43] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.
- [44] W. Mugridge, J. Hamer, and J. Hosking. Multi-methods in a statically typed programming language. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 307–324, Geneva, Switzerland, July 1991. Springer-Verlag.
- [45] D. Orleans. The Socrates programming language SourceForge project. Web site. <http://socrates-lang.sf.net/>.
- [46] 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.
- [47] D. Orleans. Incremental programming with extensible decisions. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 56–64, New York, NY, USA, 2002. ACM Press.
- [48] D. Orleans and K. J. Lieberherr. DJ: Dynamic adaptive programming in Java. In *REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 73–80, London, UK, 2001. Springer-Verlag.
- [49] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*, pages 293–323. Kluwer, 2000.
- [50] 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.

- [51] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.
- [52] H. Rajan and K. Sullivan. Classpects: Unifying aspect- and object-oriented language design. In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, pages 59–68, St. Louis, Missouri, USA, May 2005.
- [53] E. S. Ruf. *Topics in online partial evaluation*. PhD thesis, Stanford University, 1993.
- [54] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.
- [55] J. P. Secher. *Driving-based Program Transformation in Theory and Practice*. PhD thesis, Department of Computer Science, Copenhagen University, Universitetsparken 1, 2100 København Ø, July 2002.
- [56] M. H. Sørensen and R. Glück. Introduction to supercompilation. In J. Hatcliff, T. Mogensen, and P. Thiemann, editors, *Partial Evaluation: Practice and Theory*, volume 1706 of *Lecture Notes in Computer Science*, pages 246–270. Springer-Verlag, 1999.
- [57] B. Stroustrup. What is “object-oriented programming”? In G. Goos and J. Hartmann, editors, *Proc. European Conf. on Object-Oriented Programming*, Paris (France), 1987. Springer-Verlag, Lecture Notes in Computer Science no. 276.
- [58] P. Tarr and H. Ossher. *Hyper/J User and Installation Manual*. IBM T. J. Watson Research Center, Yorktown Heights, NY, USA, 2000.
- [59] P. Tarr, H. Ossher, W. Harrison, and J. Stanley M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*, pages 107–119. IEEE, May 1999.
- [60] D. B. Tucker and S. Krishnamurthi. Pointcuts and advice in higher-order languages. In M. Aksit, editor, *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003)*, pages 158–167. ACM Press, Mar. 2003.
- [61] D. B. Tucker and S. Krishnamurthi. A semantics for pointcuts and advice in higher-order languages. Technical Report CS-02-13, Brown University, March 2003.

- [62] V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, July 1986.
- [63] A. M. Ucko. Predicate dispatching in the Common Lisp Object System. Master’s thesis, Massachusetts Institute of Technology Artificial Intelligence Laboratory, Cambridge, MA, 2001.
- [64] D. Walker, S. Zdancewic, and J. Ligatti. A theory of aspects. In *ICFP ’03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 127–139, New York, NY, USA, 2003. ACM Press.
- [65] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *TOPLAS*, 26(5):890–910, September 2004. Earlier versions of this paper were presented at the 9th International Workshop on Foundations of Object-Oriented Languages, January 19, 2002, and at the Workshop on Foundations of Aspect-Oriented Languages (FOAL), April 22, 2002.
- [66] D. Weise, R. Conybeare, E. Ruf, and S. Seligman. Automatic online partial evaluation. In *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*, pages 165–191, New York, NY, USA, 1991. Springer-Verlag New York, Inc.
- [67] J. Wichman. ComposeJ: The development of a preprocessor to facilitate composition filters in the Java language. MSc. thesis, Dept. of Computer Science, University of Twente, Enschede, the Netherlands, December 1999.

Acknowledgements

The following people provided valuable comments, suggestions, and advice during the development of this research and the writing of this dissertation: Ken Anderson, Sean Barrett, Craig Chambers, Curt Clifton, Kris De Volder, Michael Ernst, Matthias Felleisen, Dan Friedman, Lars Hansen, Erik Hilsdale, Andy Latto, Karl Lieberherr, David Lorenz, Todd Millstein, Mark Nahabedian, Harold Ossher, Erik Ostrom, Johan Ovlinger, Macneil Shonle, Mike Sperber, Greg Sullivan, Mitch Wand, Galen Williamsen.

Many members of the PLT Scheme team helped out with technical support on the `plt-scheme` mailing list: Eli Barzilay, Richard Cobbe, Ryan Culpepper, John Clements, Robby Findler, Matthew Flatt, Joe Marshall, Greg Pettyjohn, Adam Wick.

The work of many previous researchers influenced and inspired this research: Harold Abelson, Edsger Dijkstra, Robert Filman, Jan Hanneman, Gregor Kiczales, Gary Leavens, Crista Lopes, Mira Mezini, Jonathan Rees, Erik Ruf, Olin Shivers, Gerald Sussman, Aaron Ucko.

Thanks to all.

Colophon

This dissertation was typeset with `SI \LaTeX` , a Scheme program included in the MzScheme distribution that extends `L \LaTeX` with support for typesetting of Scheme program fragments. All default settings were used, except the font for constants (e.g. the names of primitive procedures) was changed from `sans serif` to `typewriter`. Much of the style and formatting of the Socrates language reference chapters was inspired by the MzScheme reference manual [23].