This paper appears in ECOOP '98, the 12th European Conference on Object-Oriented Programming, Brussels, Belgium, July 20-24, 1998, pp. 186-211.

Predicate Dispatching: A Unified Theory of Dispatch

Michael Ernst, Craig Kaplan, and Craig Chambers

Department of Computer Science and Engineering
University of Washington
Seattle, WA, USA 98195-2350
{mernst,csk,chambers}@cs.washington.edu
http://www.cs.washington.edu/research/projects/cecil/

Abstract. *Predicate dispatching* generalizes previous method dispatch mechanisms by permitting arbitrary predicates to control method applicability and by using logical implication between predicates as the overriding relationship. The method selected to handle a message send can depend not just on the classes of the arguments, as in ordinary object-oriented dispatch, but also on the classes of subcomponents, on an argument's state, and on relationships between objects. This simple mechanism 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. This paper introduces predicate dispatching, gives motivating examples, and presents its static and dynamic semantics. An implementation of predicate dispatching is available.

1 Introduction

Many programming languages support some mechanism for dividing the body of a procedure into a set of cases, with a declarative mechanism for selecting the right case for each dynamic invocation of the procedure. Case selection can be broken down into tests for *applicability* (a case is a candidate for invocation if its guard is satisfied) and *overriding* (which selects one of the applicable cases for invocation).

Object-oriented languages use overloaded methods as the cases and generic functions (implicit or explicit) as the procedures. A method is applicable if the run-time class of the receiver argument is the same as or a subclass of the class on which the receiver is specialized. Multiple dispatching [BKK+86,Cha92] enables testing the classes of all of the arguments. One method overrides another if its specializer classes are subclasses of the other's, using either lexicographic (CLOS [Ste90]) or pointwise (Cecil [Cha93a]) ordering.

Predicate classes [Cha93b] automatically classify an object of class A as an instance of virtual subclass B (a subclass of A) whenever B's predicate (an arbitrary expression typically testing the runtime state of an object) is true. This creation of virtual class hierarchies makes method dispatching applicable even in cases where the effective class of an object may change over time. Classifiers [HHM90b] and modes [Tai93] are similar mechanisms for reclassifying an object into one of a number of subclasses based on a case-statement-like test of arbitrary boolean conditions.

Pattern matching (as in ML [MTH90]) bases applicability tests on the run-time datatype constructor tags of the arguments and their subcomponents. As with classifiers and modes, textual ordering determines overriding. Some languages, such as Haskell [HJW+92], allow arbitrary boolean guards to accompany patterns, restricting applicability. Views [Wad87] extend pattern matching to abstract data types by enabling them to offer interfaces like various concrete datatypes.

Predicate dispatching integrates, generalizes, and provides a uniform interface to these similar but previously incomparable mechanisms. A method declaration specifies its applicability via a *predicate expression*, which is a logical formula over class tests (i.e., tests that an object is of a particular class or one of its subclasses) and arbitrary boolean-valued expressions from the underlying programming language. A method is applicable when its predicate expression evaluates to *true*. Method m_1 overrides method m_2 when m_1's predicate logically implies that of m_2; this relationship is computed at compile time. Static typechecking verifies that, for all possible combinations of arguments to a generic function, there is always a single most-specific applicable method. This ensures that there are no "message not understood" errors (called "match not exhaustive" in ML) or "message ambiguous" errors at run-time.

Predicate expressions capture the basic primitive mechanisms underlying a wide range of declarative dispatching mechanisms. Combining these primitives in an orthogonal and general manner enables new sorts of dispatching that are not expressible by previous dispatch mechanisms. Predicate dispatching preserves several desirable properties from its object-oriented heritage, including that methods can be declared in any order and that new methods can be added to existing generic functions without modifying the existing methods or clients; these properties are not shared by pattern-matching-based mechanisms.

Section 2 introduces the syntax, semantics, and use of predicate dispatching through a series of examples. Section 3 defines its dynamic and static semantics formally. Section 4 discusses predicate tautology testing, which is the key mechanism required by the dynamic and static semantics. Section 5 surveys related work. Section 6 concludes with a discussion of future directions for research.

2 Overview

This section demonstrates some of the capabilities of predicate dispatching by way of a series of examples. We incrementally present a high-level syntax which appears in full in Fig. 6; Fig. 1 lists supporting syntactic domains. Words and symbols in **boldface** represent terminals. Angle brackets denote zero or more comma-separated repetitions of an item. Square brackets contain optional expressions.

E	$\in$ *expr*	The set of expressions in the underlying programming language
Body	$\in$ *method-body*	The set of method bodies in the underlying programming language
T	$\in$ *type*	The set of types in the underlying programming language
c	$\in$ *class-id*	The namespace of classes
m	$\in$ *method-id*	The namespace of methods and fields
f	$\in$ *field-id*	The namespace of methods and fields
p	$\in$ *pred-id*	The namespace of predicate abstractions
v, w	$\in$ *var-id*	The namespace of variables

Fig. 1. Syntactic domains and variables. Method and field names appear in the same namespace; the *method-id* or *field-id* name is chosen for clarity in the text.

Predicate dispatching is parameterized by the syntax and semantics of the host programming language in which predicate dispatching is embedded. The ideas of predicate dispatching are independent of the host language; this paper specifies only a predicate dispatching sublanguage, with *expr* the generic nonterminal for expressions in the host language. Types and signatures are used when the host language is statically typed and omitted when it is dynamically typed.

2.1 Dynamic dispatch

Each method implementation has a predicate expression which specifies when the method is applicable. Class tests are predicate expressions, as are negations, conjunctions, and disjunctions of predicate expressions.

method-sig	::=	**signature** *method-id* (⟨ *type* ⟩) : *type*	
method-decl	::=	**method** *method-id* (⟨ *formal-pattern* ⟩) [**when** *pred-expr*] *method-body*	
formal-pattern	::=	*var-id*	
pred-expr	::=	*expr* @ *class-id*	succeeds if *expr* evaluates to an instance of *class-id* or one of its subclasses
	|	**not** *pred-expr*	negation
	|	*pred-expr* **and** *pred-expr*	conjunction (short-circuited)
	|	*pred-expr* **or** *pred-expr*	disjunction (short-circuited)

Predicate expressions are evaluated in an environment with the method's formal arguments bound (see Sect. 3 for details). An omitted **when** clause indicates that the method handles all (type-correct) arguments.

Method signature declarations give the type signature shared by a family of method implementations in a generic function. A message send expression need examine only the corresponding method signature declaration to determine its type-correctness, while a set of overloaded method implementations must completely and unambiguously implement the corresponding signature in order to be type-correct.

Predicate dispatching can simulate both singly- and multiply-dispatched methods by *specializing* formal parameters on a class (via the "@*class-id*" syntax).

Specialization limits the applicability of a method to objects that are instances of the given class or one of its subclasses. More generally, predicate dispatching supports the construction of arbitrary conjunctions, disjunctions, and negations of class tests. The following example uses predicate dispatching to implement the `Zip` function which converts a pair of lists into a list of pairs:[1]

```
type List;
  class Cons subtypes List { head:Any, tail:List };
  class Nil subtypes List;

signature Zip(List, List):List;
method Zip(l1, l2) when l1@Cons and l2@Cons {
    return Cons(Pair(l1.head, l2.head), Zip(l1.tail, l2.tail)); }
method Zip(l1, l2) when l1@Nil or l2@Nil { return Nil; }
```

The first `Zip` method applies when both of its arguments are instances of `Cons` (or some subclass). The second `Zip` method uses disjunction to test whether either argument is an instance of `Nil` (or some subclass). The type checker can verify statically that the two implementations of `Zip` are mutually exclusive and exhaustive over all possible arguments that match the signature, ensuring that there will be no "message not understood" or "message ambiguous" errors at run-time, without requiring the cases to be put in any particular order.

There are several unsatisfactory alternatives to the use of implication to determine overriding relationships. ML-style pattern matching requires all cases to be written in one place and put in a particular total order, resolving ambiguities in favor of the first successfully matching pattern. Likewise, a lexicographic ordering for multimethods [Ste90] is error-prone and unnatural, and programmers are not warned of potential ambiguities. In a traditional (singly- or multiply-dispatched) object-oriented language without the ability to order cases, either the base case of `Zip` must be written as the default case for all pairs of `List` objects (unnaturally, and unsafely in the face of future additions of new subclasses of `List`), or *three* separate but identical base methods must be written: one for `Nil`×`Any`, one for `Any`×`Nil`, and a third for `Nil`×`Nil` to resolve the ambiguity between the first two. In our experience with object-oriented languages (using a pointwise, not lexicographic, ordering), these triplicate base methods for binary messages occur frequently.

As a syntactic convenience, class tests can be written in the formal argument list:

formal-pattern ::= [*var-id*] [@ *class-id*]   like *var-id* @ *class-id* in *pred-expr*

The class name can be omitted if the argument is not dispatched upon, and the formal name can be omitted if the argument is not used elsewhere in the predicate or method body.

The first `Zip` method above could then be rewritten as

[1] `Any` is the top class, subclassed by all other classes, and `Pair` returns an object containing its two arguments.

```
method Zip(l1@Cons, l2@Cons) {
  return Cons(Pair(l1.head, l2.head), Zip(l1.tail, l2.tail)); }
```

This form uses an implicit conjunction of class tests, like a multimethod.

2.2 Pattern matching

Predicates can test the run-time classes of components of an argument, just as pattern matching can query substructures, by suffixing the `@`*class* test with a record-like list of field names and corresponding class tests; names can be bound to field contents at the same time.

pred-expr ::= ...
    | *expr* @ *specializer*
specializer ::= *class-id* [{ ⟨ *field-pat* ⟩ }]
field-pat ::= *field-id* [= *var-id*] [@ *specializer*]

A *specializer* succeeds if all of the specified fields (or results of invoking methods named by *field-id*) satisfy their own *specializer*s, in which case the *var-id*s are bound to the field values or method results. As with *formal-pattern*, the formal name or specializer may be omitted.

Our syntax for pattern matching on records is analogous to that for creating a record: `{ x := 7, y := 22 }` creates a two-component record, binding the `x` field to 7 and the `y` field to 22, while `{ x = xval }` pattern-matches against a record containing an `x` field, binding the new variable `xval` to the contents of that field and ignoring any other fields that might be present. The similarity between the record construction and matching syntaxes follows ML. Our presentation syntax also uses curly braces in two other places: for record type specifiers (as in the declaration of the `Cons` class, above) and to delimit code blocks (as in the definitions of the `Zip` methods, above).

The following example, adapted from our implementation of an optimizing compiler, shows how a `ConstantFold` method can dispatch for binary operators whose arguments are constants and whose operator is integer addition:

```
type Expr;
  signature ConstantFold(Expr):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
```

The ability in pattern matching to test for constants of built-in types is a simple extension of class tests. In a prototype-based language, @ operates over objects as well as classes, as in "*answer* @ 42".

As with pattern matching, testing the representation of components of an object makes sense when the object and the tested components together implement a single abstraction. We do not advocate using pattern matching to test components of objects in a way that crosses natural abstraction boundaries.

2.3 Boolean expressions

To increase the expressiveness of predicate dispatching, predicates may test arbitrary boolean expressions from the underlying programming language. Additionally, names may be bound to values, for use later in the predicate expressions and in the method body. Expressions from the underlying programming language that appear in predicate expressions should have no externally observable side effects.[2]

pred-expr ::=	...	
	| **test** *expr*	succeeds if *expr* evaluates to *true*
	| **let** *var-id* := *expr*	evaluate *expr* and bind *var-id* to its value; always succeeds

The following extension to the `ConstantFold` example illustrates these features. Recall that in `{ value=v }`, the left-hand side is a field name and the right-hand side is a variable being bound.

```
-- Handle case of adding zero to anything (but don't be ambiguous
-- with existing method for zero plus a constant).
method ConstantFold(
          e@BinopExpr{ op@IntPlus, arg1@IntConst{ value=v }, arg2=a2 })
  when test(v == 0) and not(a2@IntConst) {
  return a2; }
method ConstantFold(
          e@BinopExpr{ op@IntPlus, arg1=a1, arg2@IntConst{ value=v } })
  when test(v == 0) and not(a1@IntConst) {
  return a1; }

... -- other special cases for operations on 0,1 here
```

[2] We do not presently enforce this restriction, but there is no guarantee regarding in what order or how many times predicate expressions are evaluated.

2.4 Predicate abstractions

Named predicate abstractions can factor out recurring tests and give names to semantically meaningful concepts in the application domain. Named predicates abstract over both tests and variable bindings — the two capabilities of inline predicate expressions — by both succeeding or failing and returning a record-like set of bindings. These bindings resemble the fields of a record or class, and similar support is given to pattern matching against a subset of the results of a named predicate invocation. Predicate abstractions thus can act like views or virtual subclasses of some object (or tuple of objects), with the results of predicate abstractions acting like the virtual fields of the virtual class. If the properties of an object tested by a collection of predicates are mutable, the object may be given different virtual subclass bindings at different times in its life, providing the benefits of using classes to organize code even in situations where an object's "class" is not fixed.

pred-sig	::=	**predsignature** *pred-id* ($\langle$ *type* $\rangle$) [**return** { $\langle$ *field-id* **:** *type* $\rangle$ }]	
pred-decl	::=	**predicate** *pred-id* ($\langle$ *formal-pattern* $\rangle$) [**when** *pred-expr*] [**return** { $\langle$ *field-id* **:=** *expr* $\rangle$ }]	
pred-expr	::=	...	
	|	*pred-id* ($\langle$ *expr* $\rangle$) [**=>** { $\langle$ *field-pat* $\rangle$ }]	test predicate abstraction
specializer	::=	*class-spec* [{ $\langle$ *field-pat* $\rangle$ }]	
class-spec	::=	*class-id*	*expr* @ *class-id* is a class test
	|	*pred-id*	*expr* @ *pred-id* [{ ... }] is alternate syntax for *pred-id*(*expr*) [**=>** { ... }]

A predicate abstraction takes a list of arguments and succeeds or fails as determined by its own predicate expression. A succeeding predicate abstraction invocation can expose bindings of names to values it computed during its evaluation, and the caller can retrieve any subset of the predicate abstraction's result bindings. Predicate signatures specify the type interface used in typechecking predicate abstraction callers and implementations. In this presentation, we prohibit recursive predicates.

Simple predicate abstractions can be used just like ordinary classes:

```
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
```

In the following example, taken from our compiler implementation, `CFG_2succ` is a control flow graph (CFG) node with two successors. Each successor is marked with whether it is a loop exit (information which, in our implementation, is dynamically maintained when the CFG is modified) and the innermost loop it does not exit. It is advantageous for an iterative dataflow algorithm to propagate values along the loop exit only after reaching a fixed point within the loop; such an algorithm would dispatch on the `LoopExit` predicate. Similarly, the algorithm could switch from iterative to non-iterative mode when exiting the outermost loop, as indicated by `TopLevelLoopExit`.

```
predsignature LoopExit(CFGnode)
  return { loop:CFGloop };
predicate LoopExit(n@CFG_2succ{ next_true = t, next_false = f })
  when test(t.is_loop_exit) or test(f.is_loop_exit)
  return { loop := outermost(t.containing_loop, f.containing_loop) };
predicate TopLevelLoopExit(n@LoopExit{ loop@TopLevelScope });
```

Only one definition per predicate abstraction is permitted; App. B relaxes this restriction.

Because object identity is not affected by these different views on an object, named predicate abstractions are more flexible than coercions in environments with side-effects. Additionally, a single object can be classified in multiple independent ways by different predicate abstractions without being forced to define all the possible conjunctions of independent predicates as explicit classes, relieving some of the problems associated with a mix-in style of class organization [HHM90b,HHM90a].

2.5 Classifiers

Classifiers [HHM90b] are a convenient syntax for imposing a linear ordering on a collection of predicates, ensuring mutual exclusion. They combine the state testing of predicate classes and the total ordering of pattern matching. An optional `otherwise` case, which executes if none of the predicates in the classifier evaluates to true, adds the guarantee of complete coverage. Multiple independent classifications of a particular class or object do not interfere with one another.

classifier-decl ::= **classify** (⟨ *formal-pattern* ⟩)
  ⟨ as *pred-id* **when** *pred-expr* [**return** { ⟨ *field-id* := *expr* ⟩ }] ⟩
  [as *pred-id* **otherwise** [**return** { ⟨ *field-id* := *expr* ⟩ }]]

Here is an example of the use of classifiers:

```
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
```

method-sig	::=	**signature** *method-id* (⟨ *type* ⟩) **:** *type*	
method-decl	::=	**method** *method-id* (⟨ *var-id* ⟩) **when** *pred-expr* *method-body*	
pred-sig	::=	**predsignature** *pred-id* (⟨ *type* ⟩) **return** **{** ⟨ *field-id* **:** *type* ⟩ **}**	
pred-decl	::=	**predicate** *pred-id* (⟨ *var-id* ⟩) **when** *pred-expr* **return** **{** ⟨ *field-id* **:=** *expr* ⟩ **}**	
$P, Q \in$ *pred-expr*	::=	*var-id* **@** *class-id*	succeeds if *var-id* is an instance of *class-id* or a subclass
	|	**test** *var-id*	succeeds if *var-id*'s value is *true*
	|	**let** *var-id* **:=** *expr*	evaluate *expr* and bind *var-id* to that value; always succeeds
	|	*pred-id* (⟨ *var-id* ⟩) **=>** **{** ⟨ *field-id* **=** *var-id* ⟩ **}**	test predicate abstraction
	|	**true**	always succeeds
	|	**not** *pred-expr*	negation
	|	*pred-expr* **and** *pred-expr*	conjunction (short-circuited)
	|	*pred-expr* **or** *pred-expr*	disjunction (short-circuited)

Fig. 2. Abstract syntax of the core language. Words and symbols in **boldface** represent terminals. Angle brackets denote zero or more comma-separated repetitions of an item. Square brackets contain optional expressions. The text uses parentheses around *pred-exprs* to indicate order of operations. Each predicate may be defined only once (App. B relaxes this restriction), and recursive predicates are forbidden.

Classifiers introduce no new primitives, but provide syntactic support for a common programming idiom. To force the classification to be mutually exclusive, each case is transformed into a predicate which includes the negation of the disjunction of all previous predicates (for details, see App. A). Therefore, an object is classified by some case only when it cannot be classified by any earlier case.

3 Dynamic and static semantics

The rest of this paper formalizes the dynamic and static semantics of a core predicate dispatching sublanguage. Figure 2 presents the abstract syntax of the core sublanguage which is used throughout this section. Appendix A defines desugaring rules that translate the high-level syntax of Fig. 6 into the core syntax.

In the remainder of this paper, we assume that all variable names are distinct so that the semantic rules can ignore the details of avoiding variable capture.

3.1 Dynamic semantics

This section explains how to select the most-specific applicable method at each message send. This selection relies on two key tests on predicated methods: whether a method is applicable to a call, and whether one method overrides another.

$\alpha, \beta \in value$	Values in the underlying programming language
$b \quad \in \{true, false\}$	Mathematical booleans
$K \quad \in (var\text{-}id \rightarrow value)$	Environments mapping variables to values
$\quad \cup (pred\text{-}id \rightarrow pred\text{-}decl)$	and predicate names to predicate declarations

$\text{lookup}(v, K) \rightarrow \alpha$	Look up variable v in environment K, returning the value α.
$K[v := \alpha] \rightarrow K'$	Bind name v to value α in environment K, resulting in the new environment K'. Any existing binding for v is overridden.
$\text{eval}(E, K) \rightarrow \alpha$	Evaluate expression E in environment K, returning the value α.
$\text{instanceof}(\alpha, c) \rightarrow b$	Determine whether value α is an instance of c or a subclass of c.
$\text{accept}(\alpha) \rightarrow b$	Coerce arbitrary program values to *true* or *false*, for use with **test**.

Fig. 3. Dynamic semantics domains and helper functions. Evaluation rules appear in Fig. 4. The host programming language supplies functions eval, instanceof, and accept.

A method is applicable if its predicate evaluates to *true*. Predicate evaluation also provides an extended environment in which the method's body is executed. Bindings created via **let** in a predicate may be used in a method body, predicate **return** clause, or the second conjunct of a conjunction whose first conjunct created the binding. Such bindings permit reuse of values without recomputation, as well as simplifying and clarifying the code. Figures 3 and 4 define the execution model of predicate evaluation.

Predicate dispatching considers one method m_1 to override another method m_2 exactly when m_1's predicate implies m_2's predicate and not vice versa. Section 4 describes how to compute the overriding relation, which can be performed at compile time.

Given the evaluation model for predicate expressions and the ability to compare predicate expressions for overriding, the execution of generic function invocations is straightforward. Suppose that generic function m is defined with the following cases:

$$\begin{array}{l}
\textbf{method}\, m(v_1, \ldots, v_n) \;\; \textbf{when}\; P_1 \;\; Body_1 \\
\textbf{method}\, m(v_1, \ldots, v_n) \;\; \textbf{when}\; P_2 \;\; Body_2 \\
\vdots \\
\textbf{method}\, m(v_1, \ldots, v_n) \;\; \textbf{when}\; P_k \;\; Body_k
\end{array}$$

To evaluate the invocation $m(E_1, \ldots, E_n)$ in the environment K, first obtain $\alpha_i = \text{eval}(E_i, K)$ for all $i = 1, \ldots, n$. Then, for $j = 1, \ldots, k$, obtain a truth value b_j and a new environment K_j through $\langle P_j, K[v_1 := \alpha_1, \ldots, v_n := \alpha_n] \rangle \Rightarrow \langle b_j, K_j \rangle$, as in the predicate invocation rules of Fig. 4.[3]

Now let I be the set of integers i such that $b_i = true$, and find $i_0 \in I$ such that P_{i_0} overrides all others in $\{P_i\}_{i \in I}$. The result of evaluating $m(E_1, \ldots, E_n)$

[3] Since we assume that all variable names are distinct and disallow lexically nested predicate abstractions, we can safely use the dynamic environment at the call site instead of preserving the static environment at the predicate abstraction's definition point.

$$\frac{}{\langle \textbf{true}, K\rangle \Rightarrow \langle \mathit{true}, K\rangle}$$

$$\frac{\text{lookup}(v, K) = \alpha \qquad \text{instanceof}(\alpha, c) = b}{\langle v \texttt{ @ } c, K\rangle \Rightarrow \langle b, K\rangle}$$

$$\frac{\text{lookup}(v, K) = \alpha \qquad \text{accept}(\alpha) = b}{\langle \textbf{test}\ \ v, K\rangle \Rightarrow \langle b, K\rangle}$$

$$\frac{\text{eval}(E, K) = \alpha \qquad K[v := \alpha] = K'}{\langle \textbf{let}\ \ v \texttt{ := } E, K\rangle \Rightarrow \langle \mathit{true}, K'\rangle}$$

$$\frac{\begin{array}{c}\forall i \in \{1, \ldots, n\} \quad \text{lookup}(v_i, K) = \alpha_i \\ \text{lookup}(p, K) = \textbf{predicate}\, p(v_1', \ldots, v_n')\ \textbf{when}\ \ P\, \textbf{return}\ \ \texttt{\{}f_1 \texttt{ := } w_1', \ldots, f_m \texttt{ := } w_m', \ldots\texttt{\}} \\ \langle P, K[v_1' := \alpha_1, \ldots, v_n' := \alpha_n]\rangle \Rightarrow \langle \mathit{false}, K'\rangle\end{array}}{\langle p(v_1, \ldots, v_n) \texttt{ => } \texttt{\{}f_1 = w_1, \ldots, f_m = w_m\texttt{\}}, K\rangle \Rightarrow \langle \mathit{false}, K\rangle}$$

$$\frac{\begin{array}{c}\forall i \in \{1, \ldots, n\} \quad \text{lookup}(v_i, K) = \alpha_i \\ \text{lookup}(p, K) = \textbf{predicate}\, p(v_1', \ldots, v_n')\ \textbf{when}\ \ P\, \textbf{return}\ \ \texttt{\{}f_1 \texttt{ := } w_1', \ldots, f_m \texttt{ := } w_m', \ldots\texttt{\}} \\ \langle P, K[v_1' := \alpha_1, \ldots, v_n' := \alpha_n]\rangle \Rightarrow \langle \mathit{true}, K'\rangle \\ \forall i \in \{1, \ldots, m\} \quad \text{lookup}(w_i', K') = \beta_i \\ K[w_1 := \beta_1, \ldots, w_m := \beta_m] = K'' \qquad\qquad (*)\end{array}}{\langle p(v_1, \ldots, v_n) \texttt{ => } \texttt{\{}f_1 = w_1, \ldots, f_m = w_m\texttt{\}}, K\rangle \Rightarrow \langle \mathit{true}, K''\rangle}$$

$$\frac{\langle P, K\rangle \Rightarrow \langle b, K'\rangle}{\langle \textbf{not}\, P, K\rangle \Rightarrow \langle \neg b, K\rangle}$$

$$\frac{\langle P, K\rangle \Rightarrow \langle \mathit{false}, K'\rangle}{\langle P\ \textbf{and}\ Q, K\rangle \Rightarrow \langle \mathit{false}, K\rangle}$$

$$\frac{\langle P, K\rangle \Rightarrow \langle \mathit{true}, K'\rangle \qquad \langle Q, K'\rangle \Rightarrow \langle \mathit{false}, K''\rangle}{\langle P\ \textbf{and}\ Q, K\rangle \Rightarrow \langle \mathit{false}, K\rangle}$$

$$\frac{\langle P, K\rangle \Rightarrow \langle \mathit{true}, K'\rangle \qquad \langle Q, K'\rangle \Rightarrow \langle \mathit{true}, K''\rangle}{\langle P\ \textbf{and}\ Q, K\rangle \Rightarrow \langle \mathit{true}, K''\rangle}$$

$$\frac{\langle P, K\rangle \Rightarrow \langle \mathit{true}, K'\rangle}{\langle P\ \textbf{or}\ Q, K\rangle \Rightarrow \langle \mathit{true}, K\rangle}$$

$$\frac{\langle P, K\rangle \Rightarrow \langle \mathit{false}, K'\rangle \qquad \langle Q, K\rangle \Rightarrow \langle \mathit{true}, K''\rangle}{\langle P\ \textbf{or}\ Q, K\rangle \Rightarrow \langle \mathit{true}, K\rangle}$$

$$\frac{\langle P, K\rangle \Rightarrow \langle \mathit{false}, K'\rangle \qquad \langle Q, K\rangle \Rightarrow \langle \mathit{false}, K''\rangle}{\langle P\ \textbf{or}\ Q, K\rangle \Rightarrow \langle \mathit{false}, K\rangle}$$

Fig. 4. Dynamic semantics evaluation rules. Domains and helper functions appear in Fig. 3. We say $\langle P, K\rangle \Rightarrow \langle b, K'\rangle$ when the predicate P evaluates in the environment K to the boolean result b, producing the new environment K'. If the result b is *false*, then the resulting environment K' is ignored. Bindings do not escape from **not** or **or** constructs; App. B relaxes the latter restriction. The starred hypothesis uses K, not K', to construct the result environment K'' because only the bindings specified in the **return** clause, not all bindings in the predicate's **when** clause, are exposed at the call site.

is then the result of evaluating $Body_{i_0}$ in the environment K_{i_0}, so that variables bound in the predicate can be referred to in the body. If no such i_0 exists, then an exception is raised: a "message not understood" error if I is empty, or a "message ambiguous" error if there is no unique most specific element of I.

An implementation can make a number of improvements to this base algorithm. Here we briefly mention just a few such optimizations. First, common subexpression elimination over predicate expressions can limit the computation done in evaluating guards. Second, precomputed implication relationships can prevent the necessity for evaluating every predicate expression. If a more specific one is true, then the less specific one is certain to be satisfied; however, such satisfaction is irrelevant since the more specific predicate will be chosen. Third, clauses and methods can be reordered to succeed or fail more quickly, as in some Prolog implementations [Zel93].

3.2 Static semantics and typechecking

The operational model of predicate dispatch described in Sect. 3.1 can raise a run-time exception at a message send if no method is applicable or if no applicable method overrides all the others. We extend the typechecking rules of the underlying language to guarantee that no such exception occurs.

Figure 5 presents the static semantic domains, helper functions, and typechecking rules for the core predicate dispatching sublanguage. The return type for a predicate invocation is an unordered record. Bindings do not escape from **not** or **or** constructs (App. B makes bindings on both sides of a **or** disjunct visible outside the disjunct).

We can separate typechecking into two parts: *client-side*, which handles all checking of expressions in the underlying language and uses method signatures to typecheck message sends, and *implementation-side*, which checks method and predicate implementations against their corresponding signatures. Only implementation-side checking is affected by predicate dispatching.

Implementation-side typechecking must guarantee *completeness* and *uniqueness*. Completeness guarantees that no "message not understood" error is raised: for every possible set of arguments at each call site, some method is applicable. Let P_m be the disjunction of the predicates of all of m's implementations, and let P_s be a predicate expressing the set of argument classes that conform to the types in the method signature. (See below for the details of predicate P_s; a class c conforms to a type T if every object which is an instance of that class has type T or a subtype of T.) If P_s implies P_m, then some method is always applicable. Uniqueness guarantees that no "message ambiguous" error is raised: for no possible set of arguments at any call site are there multiple most-specific methods. Uniqueness is guaranteed if, for each pair of predicates P and Q attached to two different implementations, either P and Q are disjoint (so their associated methods can never be simultaneously applicable) or one of the predicates implies the other (so one of the methods overrides the other). Section 4 presents implication and disjointness tests over predicate expressions.

$T \leq T'$	Type T is a subtype of T'.
$\mathit{conformant\text{-}type}(T, c)$	Return the most-specific (with respect to the subtyping partial order) type T' such that every subclass c' of c that conforms to T also conforms to T'. This helper function is supplied by the underlying programming language.
$\Gamma + \Gamma' = \Gamma''$	Overriding extension of typing environments. For each $v \in \mathrm{dom}(\Gamma')$, if $\Gamma' \models v : T'$, then $\Gamma'' \models v : T'$; for each $v \in \mathrm{dom}(\Gamma) \setminus \mathrm{dom}(\Gamma')$, if $\Gamma \models v : T$, then $\Gamma'' \models v : T$.

$$\frac{}{\Gamma \vdash \textbf{signature } m(T_1, \ldots, T_n)\,\textbf{:}\,T_r \Rightarrow \Gamma + \{m : (T_1, \ldots, T_n) \rightarrow T_r\}}$$

$$\frac{\begin{array}{c}\Gamma \models m : (T_1, \ldots, T_n) \rightarrow T_r \\ \Gamma + \{v_1 : T_1, \ldots, v_n : T_n\} \vdash P \Rightarrow \Gamma' \qquad \Gamma' \models \mathit{Body} : T_b \qquad T_b \leq T_r\end{array}}{\Gamma \vdash \textbf{method } m(v_1, \ldots, v_n) \ \textbf{ when } \ P \ \mathit{Body} \Rightarrow \Gamma}$$

$$\frac{}{\begin{array}{l}\langle \Gamma, \textbf{predsignature } p(T_1, \ldots, T_n) \ \textbf{ return } \{f_1 : T_1^r, \ldots, f_m : T_m^r\}\rangle \\ \quad \Rightarrow \Gamma + \{p : (T_1, \ldots, T_n) \rightarrow \{f_1 : T_1^r, \ldots, f_m : T_m^r\}\ \}\end{array}}$$

$$\frac{\begin{array}{c}\Gamma \models p : (T_1, \ldots, T_n) \rightarrow \{f_1 : T_1^r, \ldots, f_m : T_m^r, \ldots\} \\ \Gamma + \{v_1 : T_1, \ldots, v_n : T_n\} \vdash P \Rightarrow \Gamma' \\ \forall i \in \{1, \ldots, m\} \quad \Gamma' \models v_i' : T_i' \ \wedge \ T_i' \leq T_i^r\end{array}}{\Gamma \vdash \textbf{predicate } p(v_1, \ldots, v_n) \ \textbf{ when } \ P \ \textbf{ return } \{f_1 := v_1', \ldots, f_m := v_m'\} \Rightarrow \Gamma}$$

$$\frac{}{\Gamma \vdash \textbf{true} \Rightarrow \Gamma}$$

$$\frac{\Gamma \models v : T \qquad \mathit{conformant\text{-}type}(c, T) = T'}{\Gamma \vdash v \mathbin{@} c \Rightarrow \Gamma + \{v : T'\}}$$

$$\frac{\Gamma \models v : \mathit{Bool}}{\Gamma \vdash \textbf{test } \ v \Rightarrow \Gamma}$$

$$\frac{\Gamma \models \mathit{expr} : T}{\Gamma \vdash \textbf{let } \ v := \mathit{expr} \Rightarrow \Gamma + \{v : T\}}$$

$$\frac{\begin{array}{c}\Gamma \models p : (T_1, \ldots, T_n) \rightarrow \{f_1 : T_1^r, \ldots, f_m : T_m^r, \ldots\} \\ \Gamma \models v_1 : T_1' \quad \ldots \quad \Gamma \models v_n : T_n' \qquad T_1' \leq T_1 \quad \ldots \quad T_n' \leq T_n\end{array}}{\Gamma \vdash p(v_1, \ldots, v_n) \Rightarrow \{f_1 = v_1', \ldots, f_m = v_m'\} \Rightarrow \Gamma + \{v_1' : T_1^r, \ldots, v_m' : T_m^r\}}$$

$$\frac{\Gamma \vdash P \Rightarrow \Gamma'}{\Gamma \vdash \textbf{not } P \Rightarrow \Gamma}$$

$$\frac{\Gamma \vdash P_1 \Rightarrow \Gamma' \qquad \Gamma' \vdash P_2 \Rightarrow \Gamma''}{\Gamma \vdash P_1 \textbf{ and } P_2 \Rightarrow \Gamma''}$$

$$\frac{\Gamma \vdash P_1 \Rightarrow \Gamma' \qquad \Gamma \vdash P_2 \Rightarrow \Gamma''}{\Gamma \vdash P_1 \textbf{ or } P_2 \Rightarrow \Gamma}$$

Fig. 5. Typechecking rules. The hypothesis $\Gamma \models E : T$ indicates that typechecking in typing environment Γ assigns type T to expression E. The judgment $\Gamma \vdash P \Rightarrow \Gamma'$ represents extension of typechecking environments: given type environment Γ, P typechecks and produces new typechecking environment Γ'.

Completeness checking requires a predicate P_s that expresses the set of tuples of values $v_1, \ldots, v_n$ conforming to some signature's argument types $T_1, \ldots, T_n$; this predicate depends on the host language's model of classes and typing. If classes and types are the same, and all classes are concrete, then the corresponding predicate is simply $v_1 \mathbin{@} T_1$ **and** ... **and** $v_n \mathbin{@} T_n$. If abstract classes are allowed, then each $v_i \mathbin{@} T_i$ is replaced with $v_i \mathbin{@} T_{i1}$ **or** ... **or** $v_i \mathbin{@} T_{im}$, where the T_{ij} are the top concrete subclasses of T_i. If inheritance and subtyping are separate notions, then the predicates become more complex.

Our typechecking need not test that methods conform to signatures, unlike previous work on typechecking multimethods [CL95]. In predicate dispatching, a method's formal argument has two distinct types: the "external" type derived from the signature declaration, and the possibly finer "internal" type guaranteed by successful evaluation of the method's predicate. The individual @ tests narrow the type of the tested value to the most-specific type to which all classes passing the test conform, in a host-language-specific manner, using *conformant-type*. The *conformant-type* function replaces the more complicated conformance test of earlier work.

4 Comparing predicate expressions

The static and dynamic semantics of predicate dispatching require compile-time tests of implication between predicates to determine the method overriding relationship. The static semantics also requires tests of completeness and uniqueness to ensure the absence of "message not understood" errors and "message ambiguous" errors, respectively. All of these tests reduce to tautology tests over predicates. Method m_1 with predicate p_1 overrides method m_2 with predicate p_2 iff p_1 implies p_2 — that is, if (**not** p_1) **or** p_2 is true. A set of methods is complete if the disjunction of their predicates is true. Uniqueness for a set of methods requires that for any pair of methods, either one's predicate overrides the other's, or the two predicates are logically exclusive. Two formulas are mutually exclusive exactly if one implies the negation of the other.

Section 4.1 presents a tautology test over predicate expressions which is simple, sound, and complete up to equivalence of arbitrary program expressions in **test** constructs, which we treat as black boxes. Because determining logical tautology is NP-complete, in the worst case an algorithm takes exponential time in the size of the predicate expressions. For object-oriented dispatch, this is the number of arguments to a method (a small constant). Simple optimizations (Sect. 4.2) make the tests fast in many practical situations. This cost is incurred only at compile time; at run time, precomputed overriding relations among methods are simply looked up.

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.

4.1 The base algorithm

The base algorithm for testing predicate tautology has three components. First, the predicate expression is canonicalized to macro-expand predicate abstractions, eliminate variable bindings, and use canonical names for formal arguments. This transformation prevents different names for the same value from being considered distinct. Second, implication relations are computed among the atomic predicates (for instance, x `@ int` implies x `@ num`). Finally, the canonicalized predicate is tested for every assignment of atomic predicates to truth values which is consistent with the atomic predicate implications. The predicate is a tautology iff evaluating it in every consistent truth assignment yields *true*.

Canonicalization Canonicalization performs the following transformations:

- Expand predicate calls inline, replacing the **=>** clause by a series of **let** bindings.
- Replace **let**-bound variables by the expressions to which they are bound, and replace **let** expressions by **true**.
- Canonically rename formal parameters according to their position in the formal list.

After canonicalization, each predicate expression is a logical formula over the following atoms with connectives **and**, **or**, and **not**.

pred-atom ::= **true**
  | **test** *expr*
  | *expr* @ *class-id*

Canonicalized predicates are a compile-time construct used only for predicate comparison; they are never executed. Canonicalized predicates bind no variables, and they use only global variables and formal parameters.

In the worst case, canonicalization exponentially blows up expression sizes. For instance, in

$$\textbf{let}\, x_1 = x{+}x\ \textbf{and}\,\textbf{let}\, x_2 = x_1{+}x_1\ \textbf{and}\,\textbf{let}\, x_3 = x_2{+}x_2\ \textbf{and}\ldots\textbf{and}\,\textbf{test}\ x_n = y\ ,$$

the final x_n is replaced by an expression containing 2^n instances of x. Inline expansion of predicate abstractions similarly contributes to this blowup. As with ML typechecking [KM89], which is exponential in the worst case but linear in practice, we anticipate that predicates leading to exponential behavior will be rare.

In what follows, we consider two expressions identical if, after canonicalization, they have the same abstract syntax tree.

Omitting the canonicalization step prevents some equivalent expressions from being recognized as such, but does not prevent the remainder of the algorithm from succeeding when results are named and reused rather than the computation repeated.

Truth assignment checking We present a simple exponential-time algorithm to check logical tautology; because the problem is NP-complete, any algorithm takes exponential time in the worst case. Let there be n distinct predicate atoms in the predicate; there are 2^n different truth assignments for those atoms. Not all of those truth assignments are consistent with the implications over predicate atoms: for instance, it is not sensible to set `a @ int` to *true* but `a @ num` to *false*, because `a @ int` implies `a @ num`. If every consistent truth assignment satisfies the predicate, then the predicate is a tautology. Each check of a single truth assignment takes time linear in the size of the predicate expression, for a total time of $O(n2^n)$.

The following rules specify implication over (possibly negated) canonical predicate atoms.

$$
\begin{aligned}
E_1 \mathbin{@} c_1 \Rightarrow E_2 \mathbin{@} c_2 &\quad \text{iff} \quad (E_1 \equiv E_2) \text{ and } (c_1 \text{ is a subclass of } c_2) \\
E_1 \mathbin{@} c_1 \Rightarrow \mathbf{not}(E_2 \mathbin{@} c_2) &\quad \text{iff} \quad (E_1 \equiv E_2) \text{ and } (c_1 \text{ is disjoint from } c_2) \\
a_1 \Rightarrow a_2 &\quad \text{iff} \quad \mathbf{not}\, a_2 \Rightarrow \mathbf{not}\, a_1 \\
a_1 \Rightarrow \mathbf{not}\, a_2 &\quad \text{iff} \quad a_2 \Rightarrow \mathbf{not}\, a_1
\end{aligned}
$$

Two classes are disjoint if they have no common descendant, and $\mathbf{not}\,\mathbf{not}\, a = a$.

4.2 Optimizations

The worst-case exponential-time cost to check predicate tautology need not prevent its use in practice. Satisfiability is checked only at compile time. When computing overriding relationships, the predicates tend to be small (linear in the number of arguments to a method). We present heuristics that reduce the costs even further.

Logical simplification — such as eliminating uses of **true**, **false**, a **and not** a, and a **or not** a, and replacing **not not** a by a — can be performed as part of canonicalization to reduce the size of predicate expressions.

Unrelated atomic predicates can be treated separately. To determine whether **method** $m_1(f_1@c_1, f_2@c_2)\{...\}$ overrides **method** $m_1(f_1@c_3, f_2@c_4)\{...\}$ it is sufficient to independently determine the relationship between c_1 and c_3 and that between c_2 and c_4. Two tests with a smaller exponent replace one with a larger one, substantially reducing the overall cost. This technique always solves ordinary single and multiple dispatching overriding in time constant and linear in the number of formals, respectively, by examining each formal position independently. The technique also applies to more complicated cases, by examining subsets of formal parameters which appear together in tests from the underlying programming language.

It is not always necessary to completely expand predicate abstraction calls as part of canonicalization. If relations between predicate abstractions or other predicate expressions are known, then the tautology test can use them directly. As one example, different cases of a classifier are mutually exclusive by definition.

The side conditions on atomic predicate values (their implication relationships) usually prevent the need to check all 2^n different truth assignments for a predicate containing n atomic predicates. When `a @ int` is set to *true*, then all truth assignments which set `a @ num` to *false* can be skipped without further consideration.

Finally, it may be possible to achieve faster results in some cases by recasting the tautology test. Rather than attempting to prove that every truth assignment satisfies a predicate expression, it may be advantageous to search for a single truth assignment that satisfies its negation.

5 Related work

5.1 Object-oriented approaches

In the model of predicate dispatching, traditional object-oriented dispatching translates to either a single class test on the receiver argument or, for multiple dispatching, a conjunction of class tests over several arguments. Full predicate dispatching additionally enables testing arbitrary boolean expressions from the underlying language, accessing and naming subcomponents of the arguments, performing tests over multiple arguments, and arbitrarily combining tests via conjunction, disjunction, and negation. Also, named predicate abstractions effectively introduce new virtual classes and corresponding subclassing links into the program inheritance hierarchy. Predicate dispatching preserves the ability in object-oriented languages to statically determine when one method overrides another and when no message lookup error can occur. Singly-dispatched object-oriented languages have efficient method lookup algorithms and separate typechecking, which depend crucially on the absence of any separate modules that dispatch on other argument positions. Multiply-dispatched object-oriented languages have more challenging problems in implementation [KR89,CTK94,AGS94] and typechecking [CL95], and predicate dispatching in its unrestricted form shares these challenges.

Predicate classes [Cha93b] are an earlier extension of object-oriented dispatching to include arbitrary boolean predicates. A predicate class which inherits from class A and has an associated predicate expression *guard* would be modeled as a named predicate abstraction that tests @A **and** *guard*. Predicate dispatching is more general, for example by being able to define predicates over multiple arguments. Predicate dispatching exploits the structure of **and**, **or**, and **not** to automatically determine when no message lookup error can occur, while typechecking of predicate classes relies on uncheckable user assertions about the relations between the predicate classes' guard expressions.

Classifiers in Kea [HHM90b,HHM90a,MHH91] let an instance of a class be dynamically reclassified as being of a subclass. A classifier for a class is composed of a sequence of predicate/subclass pairs, with an object of the input class automatically classified as being of the subclass with the first successful predicate. Because the sequence of predicates is totally ordered and the first successful predicate takes precedence over all later predicates, a classifier provides a concise syntax for a set of mutually exclusive, exhaustive predicate abstractions. Predicate abstractions are more general than classifiers in many of the ways discussed above, but they also provide syntactic support for this important idiom. Kea is a purely functional language, so classifiers do not need to consider the semantics of reclassifying objects when the values of predicates change; predicate dispatching addresses this issue by (conceptually) performing reclassification as needed as part of message dispatching.

Modes [Tai93] are another mechanism for adding dynamic reclassification of a class into a subclass. Unlike predicate classes and classifiers, the modes of a class are not first-class subclasses but rather internal components of a class that cannot be extended externally and that cannot exploit inheritance to factor shared code. Mode reselection can be done either explicitly at the end of each method or implicitly after each assignment using a declaratively specified classification.

5.2 Pattern matching approaches

Predicate dispatching supports many of the facilities found in pattern matching as in ML [MTH90] and Haskell [HJW+92], including tests over arbitrary nested structure, binding of names to subcomponents, and arbitrary boolean guard expressions. Predicate dispatching additionally supports inheritance (its class tests are more general than datatype constructor patterns), disjunctions and negations of tests and conjunctions of tests on the same object, and named predicate abstractions to factor out common patterns of tests and to offer conditional views of objects extended with virtual fields. The patterns in a function are totally ordered, while predicate dispatching computes a partial order over predicates and warns when two patterns might be ambiguous. Finally, new methods can be added to existing generic functions without changing any existing code, while new patterns can be added to a function only by modifying it.

Views [Wad87] extend pattern matching to abstract data types by allowing an abstract data type to offer a number of views of itself as a concrete datatype, over which pattern matching is defined. Predicate dispatching supports "pattern matching" over the results of methods (by **let**-binding their results to names and then testing those names, just as field contents are bound and tested), and those methods can serve as accessor functions to a virtual view of the object, for instance `rho` and `theta` methods presenting a polar view of a cartesian point. Views must be isomorphisms, which enables equational reasoning over them; by contrast, named predicate abstractions provide conditional views of an object without requiring the presence of both in and out views.

Pizza [OW97] supports both algebraic datatypes (and associated pattern matching) and object-oriented dispatching, but the two mechanisms are largely distinct. The authors argue that datatypes are good for fixed numbers of representations with extensible operations, while classes are good for a fixed set of operations with extensible representations. By integrating pattern matching and dispatching, including multimethods, predicate dispatching achieves extensibility in both dimensions along with the syntactic convenience of pattern matching. Predicate dispatching faces more difficult implementation and separate typechecking challenges with the shift to multimethod-like dispatching.

6 Conclusions

Many language features express the concept of selecting a most-specific applicable method from a collection of candidates, including object-oriented dispatch, pattern matching, views, predicate classes, and classifiers. Predicate dispatching integrates and generalizes these mechanisms in a single framework, based on a core language of boolean expressions over class tests and arbitrary expressions, explicit binding forms to generalize features of pattern matching, and named predicate abstractions with result bindings. By providing a single integrated mechanism, programs can take advantage of various styles of dispatch and even combine them to create applicability conditions that were previously either impossible or inconvenient to express.

We have implemented predicate dispatching in the context of Dubious, a simple core multiply-dispatched object-oriented programming language. The implementation supports all the examples presented in this paper, although for clarity this paper uses a slightly different presentation syntax. The implementation supports the full core language of Sect. 3 and many of the syntactic sugars of App. A. This implementation was helpful in verifying our base design. We expect that it will also provide insight into the advantages and disadvantages of programming with predicate dispatching, as well as help us to evaluate optimization strategies. The implementation is available from `http://www.cs.washington.edu/research/projects/cecil/www/Gud/`.

So far, we have focused on developing the static and dynamic semantics for predicate dispatching. Two unresolved practical issues that we will address in the future are efficient implementation techniques and separate typechecking support for predicate dispatching. We anticipate that efficient implementations of unrestricted predicate dispatching will build upon work on efficient implementation of multimethod dispatching and on predicate classes. In addition, static analyses that factor a collection of predicates to avoid redundant tests and side-effect analyses that determine when predicates need not be re-evaluated appear to be promising lines for future research. Similarly, separate typechecking of collections of predicated methods will build upon current work to develop modular and incremental methods for typechecking multimethods [CL95].

Acknowledgments

Todd Millstein, Vassily Litvinov, Wilson Hsieh, David Grove, and the anonymous referees made helpful comments on a draft of this paper. This research is supported in part by an NSF grant (number CCR-9503741), an NSF Young Investigator Award (number CCR-9457767), a grant from the Office of Naval Research (contract number N00014-94-1-1136), an IBM graduate fellowship, an FCAR graduate scholarship, and gifts from Sun Microsystems, IBM, Xerox PARC, Pure Software, and Edison Design Group.

References

[AGS94] Eric Amiel, Olivier Gruber, and Eric Simon. Optimizing multi-method dispatch using compressed dispatch tables. In *Proceedings OOPSLA '94*, pages 244–258, Portland, OR, October 1994.

[BKK+86] Daniel G. Bobrow, Ken Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. Commonloops: Merging lisp and object-oriented programming. In *Proceedings OOPSLA '86*, pages 17–29, November 1986. Published as ACM SIGPLAN Notices, volume 21, number 11.

[Cha92] Craig Chambers. Object-oriented multi-methods in Cecil. In O. Lehrmann Madsen, editor, *Proceedings ECOOP '92*, LNCS 615, pages 33–56, Utrecht, The Netherlands, June 1992. Springer-Verlag.

[Cha93a] Craig Chambers. The Cecil language: Specification and rationale. Technical Report UW-CSE-93-03-05, Department of Computer Science and Engineering. University of Washington, March 1993.

[Cha93b] Craig Chambers. Predicate classes. In O. Nierstrasz, editor, *Proceedings ECOOP '93*, LNCS 707, pages 268–296, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[CL95] Craig Chambers and Gary T. Leavens. Typechecking and modules for multi-methods. *ACM Transactions on Programming Languages and Systems*, 17(6):805–843, November 1995.

[CTK94] Weimin Chen, Volker Turau, and Wolfgang Klas. Efficient dynamic lookup strategy for multi-methods. In M. Tokoro and R. Pareschi, editors, *Proceedings ECOOP '94*, LNCS 821, pages 408–431, Bologna, Italy, July 1994. Springer-Verlag.

[HHM90a] J. Hamer, J.G. Hosking, and W.B. Mugridge. A method for integrating classification within an object-oriented environment. Technical Report Auckland Computer Science Report No. 48, Department of Computer Science, University of Auckland, October 1990.

[HHM90b] J.G. Hosking, J. Hamer, and W.B. Mugridge. Integrating functional and object-oriented programming. In *Technology of Object-Oriented Languages and Systems TOOLS 3*, pages 345–355, Sydney, 1990.

[HJW+92] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, Maria Guzman, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language Haskell, version 1.2. *SIGPLAN Notices*, 27(5), May 1992.

[KM89] Paris C. Kanellakis and John C. Mitchell. Polymorphic unification and ML typing. In ACM-SIGPLAN ACM-SIGACT, editor, *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages (POPL '89)*, pages 105–115, Austin, TX, USA, January 1989. ACM Press.

[KR89] Gregor Kiczales and Luis Rodriguez. Efficient method dispatch in PCL. Technical Report SSL 89-95, Xerox PARC Systems Sciences Laboratory, 1989.

[MHH91] Warwick B. Mugridge, John Hamer, and John G. Hosking. Multi-methods in a statically-typed programming language. In P. America, editor, *Proceedings ECOOP '91*, LNCS 512, pages 307–324, Geneva, Switzerland, July 15-19 1991. Springer-Verlag.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 146–159, January 1997.

[Ste90] Guy L. Steele Jr. *Common Lisp: The Language*. Digital Press, Bedford, MA, 1990. Second edition.

[Tai93] Antero Taivalsaari. Object-oriented programming with modes. *Journal of Object-Oriented Programming*, pages 25–32, June 1993.

[Wad87] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 307–313, Munich, Germany, January 1987.

[Zel93] John M. Zelle. Learning search-control heuristics for logic programs: Applications tospeed-up learning and languageacquisitions. Technical Report AI93-200, University of Texas, Austin, May 1, 1993.

A Desugaring rules

The following rewrite rules desugar the high-level syntax of Fig. 6 into the core abstract syntax of Fig. 2. The rules are grouped by their intention, such as providing names for arbitrary expressions or breaking down compound predicate abstractions.

For brevity, we omit the rewrite rules which introduce defaults for omitted optional program fragments: dummy variables for pattern variables, "`@Any`" specializers, empty field pattern sets in specializers, and "**when true**" and "**return { }**" clauses. Additional rules may be introduced to simplify the resulting formula, such as converting "v @ `Any`" to "**true**" and performing logical simplification.

For brevity, we use $\bigwedge_{i=1}^{n} \{P_i\}$ to stand for the conjunction of the terms: P_1 **and** ... **and** P_n. When $n = 0$, $\bigwedge_{i=1}^{n} \{P_i\}$ stands for **true**. Variables v' and v'_i are new variables which do not appear elsewhere in the program. Ceiling braces $\lceil\cdot\rceil$ surround (potentially) sugared expressions; application of the rewrite rules eliminates those braces.

A.1 Declarations

These rules move specializers from formal lists into **when** clauses.

$$\lceil \textbf{method } m(v_1 \mathbin{@} S_1, \ldots, v_n \mathbin{@} S_n) \ \textbf{ when } \ P \ \mathit{Body}\rceil$$
$$\Longrightarrow \textbf{method } m(v_1, \ldots, v_n) \ \textbf{ when } \ \textstyle\bigwedge_{i=1}^{n} \{\lceil v_i \mathbin{@} S_i\rceil\} \textbf{ and } \lceil P\rceil \ \ \mathit{Body}$$
$$\lceil \textbf{predicate } p(v_1 \mathbin{@} S_1, \ldots, v_n \mathbin{@} S_n) \ \textbf{ when } \ P \ \textbf{ return } \ \{f_1 := E_1, \ldots, f_m := E_m\}\rceil$$
$$\Longrightarrow \textbf{predicate } p(v_1, \ldots, v_n)$$
$$\textbf{when } \ \textstyle\bigwedge_{i=1}^{n} \{\lceil v_i \mathbin{@} S_i\rceil\} \textbf{ and } \ \lceil P\rceil \ \lceil \textbf{return } \{f_1 := E_1, \ldots, f_m := E_m\}\rceil$$

method-sig	::=	**signature** *method-id* (⟨ *type* ⟩) **:** *type*	
method-decl	::=	**method** *method-id* (⟨ *formal-pattern* ⟩) [**when** *pred-expr*] *method-body*	
pred-sig	::=	**predsignature** *pred-id* (⟨ *type* ⟩) [**return** **{** ⟨ *field-id* **:** *type* ⟩ **}**]	
pred-decl	::=	**predicate** *pred-id* (⟨ *formal-pattern* ⟩) [**when** *pred-expr*] [**return** **{** ⟨ *field-id* **:=** *expr* ⟩ **}**]	
classifier-decl	::=	**classify** (⟨ *formal-pattern* ⟩) ⟨ **as** *pred-id* **when** *pred-expr* [**return** **{** ⟨ *field-id* **:=** *expr* ⟩ **}**] ⟩ [**as** *pred-id* **otherwise** [**return** **{** ⟨ *field-id* **:=** *expr* ⟩ **}**]]	
$P, Q \in$ *pred-expr*	::=	*expr* **@** *specializer*	succeeds if *expr* evaluates to a value that satisfies *specializer*
	$\mid$	**test** *expr*	succeeds if *expr* evaluates to *true*
	$\mid$	**let** *var-id* **:=** *expr*	evaluate *expr* and bind *var-id* to its value; always succeeds
	$\mid$	*pred-id* (⟨ *expr* ⟩) [**=>** **{** ⟨ *field-pat* ⟩ **}**]	test predicate abstraction
	$\mid$	**true**	always succeeds
	$\mid$	**false**	never succeeds
	$\mid$	**not** *pred-expr*	negation
	$\mid$	*pred-expr* **and** *pred-expr*	conjunction (short-circuited)
	$\mid$	*pred-expr* **or** *pred-expr*	disjunction (short-circuited)
formal-pattern	::=	[*var-id*] [**@** *specializer*]	like *var-id***@***specializer* in *pred-expr*
$F \in$ *field-pat*	::=	*field-id* [**=** *var-id*] [**@** *specializer*]	
$S \in$ *specializer*	::=	*class-spec* [**{** ⟨ *field-pat* ⟩ **}**]	
$C \in$ *class-spec*	::=	*class-id*	*expr* **@** *class-id* is a class test
	$\mid$	*pred-id*	*expr* **@** *pred-id* [**{** ... **}**] is sugar for *pred-id*(*expr*) [**=>{** ... **}**]
	$\mid$	**not** *class-spec*	succeeds if *class-spec* does not
	$\mid$	*class-spec* **&** *class-spec*	succeeds if both *class-spec*s do
	$\mid$	*class-spec* $\mid$ *class-spec*	succeeds if either *class-spec* does

Fig. 6. Full extended abstract syntax for predicate dispatching. The syntax is as presented incrementally in Sect. 2, with the addition of the **true** and **false** predicate expressions and the **not**, **&**, and **|** class specializers. Words and symbols in **boldface** represent terminals. Angle brackets denote zero or more comma-separated repetitions of an item. Square brackets contain optional expressions. Each predicate may be defined only once (App. B relaxes this restriction), and recursive predicates are forbidden.

A.2 Naming of non-variable expressions

The core language permits arbitrary expressions only in **let** bindings and uses variable references elsewhere. These rules introduce **let** bindings and are intended to fire only once (alternately, only if one of the E expressions is not a mere variable reference), lest the @ and predicate application rules cause an infinite loop in desugaring.

$$\lceil E \mathbin{@} S \rceil \Longrightarrow \textbf{let}\ v' := E\ \textbf{and}\ \lceil v' \mathbin{@} S \rceil$$

$$\lceil \textbf{test } E \rceil \Longrightarrow \textbf{let } v' := E \textbf{ and test } v'$$

$$\lceil p(E_1, \ldots, E_n) \texttt{ => } \{F_1, \ldots, F_m\} \rceil$$
$$\Longrightarrow \textstyle\bigwedge_{i=1}^{n} \{\textbf{let } v'_i := E_i\} \textbf{ and } \lceil p(v'_1, \ldots, v'_n) \texttt{ => } \{F_1, \ldots, F_m\} \rceil$$

$$\lceil \textbf{return } \{f_1 := E_1, \ldots, f_m := E_m\} \rceil$$
$$\Longrightarrow \textbf{and} \textstyle\bigwedge_{i=1}^{m} \{\textbf{let } v'_i := E_i\} \textbf{ return } \{f_1 := v'_1, \ldots, f_m := v'_m\}$$

A.3 Compound predicate expressions

These rules show how to desugar **false** and compound predicate expressions.

$$\lceil \textbf{false} \rceil \Longrightarrow \textbf{not true}$$
$$\lceil \textbf{not } P \rceil \Longrightarrow \textbf{not } \lceil P \rceil$$
$$\lceil P_1 \textbf{ and } P_2 \rceil \Longrightarrow \lceil P_1 \rceil \textbf{ and } \lceil P_2 \rceil$$
$$\lceil P_1 \textbf{ or } P_2 \rceil \Longrightarrow \lceil P_1 \rceil \textbf{ or } \lceil P_2 \rceil$$

A.4 Field bindings

Pattern matching permits arbitrarily nested tests and simultaneous matching on fields of objects, fields of predicate results, and results of arbitrary method invocations. These rules separate these varieties of record patterns and flatten tests.

We introduce the concept of a class specializer *generating* a field. A class name generates the fields in the class; a predicate name generates the fields in the predicate's **return** clause; a conjunction generates the fields generated by either of its conjuncts; and a disjunction generates the fields generated by both of its disjuncts.

If F_i is generated by $C \neq c$ for $1 \leq i \leq m < n$:

$$\lceil v \text{ @ } C\ \{F_1, \ldots, F_m, \ldots, F_n\} \rceil$$
$$\Longrightarrow \lceil v \text{ @ } C\ \{F_1, \ldots, F_m\} \rceil \textbf{ and } \lceil v \text{ @ } \texttt{Any}\ \{F_{m+1}, \ldots, F_n\} \rceil$$

$$\lceil v \text{ @ } c\ \{f_1 = v_1 \text{ @ } S_1, \ldots, f_n = v_n \text{ @ } S_n\} \rceil$$
$$\Longrightarrow v \text{ @ } c \textbf{ and } \textstyle\bigwedge_{i=1}^{n} \{\textbf{let } v_i := v.f_i \textbf{ and } \lceil v_i \text{ @ } S_i \rceil\}$$

$$\lceil v \text{ @ } p\ \{f_1 = v_1 \text{ @ } S_1, \ldots, f_n = v_n \text{ @ } S_n\} \rceil$$
$$\Longrightarrow p(v) \texttt{ => } \{f_1 = v_1, \ldots, f_n = v_n\} \textbf{ and } \textstyle\bigwedge_{i=1}^{n} \{\lceil v' \text{ @ } S_i \rceil\}$$

A.5 Compound predicate abstractions

These rules simplify compound predicate abstractions.

$$\lceil v \text{ @ } \textbf{not } C\{F_1, \ldots, F_m\} \rceil \Longrightarrow \textbf{not } \lceil v \text{ @ } C\{F_1, \ldots, F_m\} \rceil$$

If F_i, $m+1 \leq i \leq n$, is generated by C_2 (& rule only):

$$\lceil v \text{ @ } C_1 \,\&\, C_2\{F_1, \ldots, F_m, \ldots, F_n\} \rceil \Longrightarrow \lceil v \text{ @ } C_1\{F_1, \ldots, F_m\} \rceil \textbf{ and } \lceil v \text{ @ } C_2\{F_{m+1}, \ldots, F_n\} \rceil$$

$$\lceil v \text{ @ } C_1 \mid C_2\{F_1, \ldots, F_m\} \rceil \Longrightarrow \lceil v \text{ @ } C_1\{F_1, \ldots, F_m\} \rceil \textbf{ or } \lceil v \text{ @ } C_2\{F_1, \ldots, F_m\} \rceil$$

A.6 Classifiers

Sequential ordering over classifier cases is enforced by creating extra predicates d_i such that d_i is true if any c_j, $j \leq i$, is true. Each c_i is true only if d_{i-1} is not (that is, no previous conjunct was true).

$$
\left\lceil
\begin{array}{l}
\textbf{classify}(v_1 \mathbin{@} S_1, \ldots, v_m \mathbin{@} S_m) \\
\quad \textbf{as } c_1 \textbf{ when } P_1 \textbf{ return } \{f_{1,1} := v'_{1,1}, \ldots, f_{1,m_1} := v'_{1,m_1}\} \\
\quad \vdots \\
\quad \textbf{as } c_n \textbf{ when } P_n \textbf{ return } \{f_{n,1} := v'_{n,1}, \ldots, f_{n,m_n} := v'_{n,m_n}\} \\
\quad \textbf{as } c_{n+1} \textbf{ otherwise return } \{f_{n+1,1} := v'_{n+1,1}, \ldots, f_{n+1,m_{n+1}} := v'_{n+1,m_{n+1}}\}
\end{array}
\right\rceil
$$

$$
\Longrightarrow
\left\lceil
\begin{array}{l}
\textbf{predicate } c_1(v_1 \mathbin{@} S_1, \ldots, v_m \mathbin{@} S_m) \textbf{ when } P_1 \\
\qquad \textbf{return } \{f_{1,1} := v'_{1,1}, \ldots, f_{1,m_1} := v'_{1,m_1}\};
\end{array}
\right\rceil
$$

$$
\lceil \textbf{predicate } d_1(v_1 \mathbin{@} S_1, \ldots, v_m \mathbin{@} S_m) \textbf{ when } P_1; \rceil
$$

$$
\left\lceil
\begin{array}{l}
\textbf{predicate } c_2(v_1 \mathbin{@} S_1, \ldots, v_m \mathbin{@} S_m) \textbf{ when } P_2 \textbf{ and not } d_1(v_1, \ldots, v_m) \\
\qquad \textbf{return } \{f_{2,1} := v'_{2,1}, \ldots, f_{2,m_2} := v'_{2,m_2}\};
\end{array}
\right\rceil
$$

$$
\lceil \textbf{predicate } d_2(v_1 \mathbin{@} S_1, \ldots, v_m \mathbin{@} S_m) \textbf{ when } d_1(v_1, \ldots, v_m) \textbf{ or } P_2; \rceil
$$

$$
\vdots
$$

$$
\left\lceil
\begin{array}{l}
\textbf{predicate } c_n(v_1 \mathbin{@} S_1, \ldots, v_m \mathbin{@} S_m) \textbf{ when } P_n \textbf{ and not } d_{n-1}(v_1, \ldots, v_m) \\
\qquad \textbf{return } \{f_{n,1} := v'_{n,1}, \ldots, f_{n,m_n} := v'_{n,m_n}\};
\end{array}
\right\rceil
$$

$$
\lceil \textbf{predicate } d_n(v_1 \mathbin{@} S_1, \ldots, v_m \mathbin{@} S_m) \textbf{ when } d_{n-1}(v_1, \ldots, v_m) \textbf{ or } P_n; \rceil
$$

$$
\left\lceil
\begin{array}{l}
\textbf{predicate } c_{n+1}(v_1 \mathbin{@} S_1, \ldots, v_m \mathbin{@} S_m) \textbf{ when not } d_n(v_1, \ldots, v_m) \\
\qquad \textbf{return } \{f_{n+1,1} := v'_{n+1,1}, \ldots, f_{n+1,m_{n+1}} := v'_{n+1,m_{n+1}}\};
\end{array}
\right\rceil
$$

B Bindings escaping "or"

In the static and dynamic semantics presented in Sect. 3, bindings never escape from **or** predicate expressions. Relaxing this constraint provides extra convenience to the programmer and permits more values to be reused rather than recomputed. It is also equivalent to permitting overloaded predicates or multiple predicate definitions — so far we have permitted only a single definition of each predicate. With appropriate variable renaming, multiple predicate definitions that rely on a dispatching-like mechanism to select the most specific applicable method can be converted into uses of **or**, and vice versa.

For example, the two `ConstantFold` methods of Sect. 2.3 can be combined into a single method. Eliminating code duplication is a prime goal of object-oriented programming, but the previous version repeated the body twice. Use of a helper method would unnecessarily separate the dispatching conditions from the code being executed, though a helper predicate could reduce code duplication in the predicate expression.

```
-- 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; }
```

As another example, the `LoopExit` example of Sect. 2.4 can be extended to present a view which indicates which branch of the `CFG_2succ` is the loop exit and which the backward branch. When performing iterative dataflow, this is the only information of interest, and in our current implementation (which uses predicate classes [Cha93b]) we generally recompute this information after discovering that an object is a `LoopExit`. Presenting a view which includes this information directly would improve the code's readability and efficiency.

```
predsignature LoopExit(CFGnode)
  return { loop:CFGloop, next_loop:CFGedge, next_exit:CFGedge };
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 };
```

Permitting bindings which appear on both sides of **or** to escape requires the following changes to the dynamic semantics of Fig. 4. (The third rule is unchanged from Fig. 4 but included here for completeness.)

$$\frac{\langle P, K\rangle \Rightarrow \langle \mathit{true}, K'\rangle}{\langle P \textbf{ or } Q, K\rangle \Rightarrow \langle \mathit{true}, K'\rangle}$$

$$\frac{\langle P, K\rangle \Rightarrow \langle \mathit{false}, K'\rangle \qquad \langle Q, K\rangle \Rightarrow \langle \mathit{true}, K''\rangle}{\langle P \textbf{ or } Q, K\rangle \Rightarrow \langle \mathit{true}, K''\rangle}$$

$$\frac{\langle P, K\rangle \Rightarrow \langle \mathit{false}, K'\rangle \qquad \langle Q, K\rangle \Rightarrow \langle \mathit{false}, K''\rangle}{\langle P \textbf{ or } Q, K\rangle \Rightarrow \langle \mathit{false}, K\rangle}$$

These execution rules do not reflect that only the bindings appearing on both sides, not all those appearing on the succeeding side, should escape; however, the typechecking rules below guarantee that only the appropriate variables are referenced.

The static semantics of Fig. 5 are modified to add two helper functions and replace a typechecking rule:

$T \sqcup T' = T''$	Least upper bound over types. T'' is the least common supertype of T and T'.
$\sqcup_{\mathrm{env}}(\Gamma, \Gamma') = \Gamma''$	Pointwise lub over typing environments. For each $v \in \mathrm{dom}(\Gamma'') = \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma')$, if $\Gamma \models v : T$ and $\Gamma' \models v : T'$, then $\Gamma'' \models v : T \sqcup T'$.

$$\frac{\Gamma \vdash P_1 \Rightarrow \Gamma' \qquad \Gamma \vdash P_2 \Rightarrow \Gamma'' \qquad \sqcup_{\mathrm{env}}(\Gamma', \Gamma'') = \Gamma'''}{\Gamma \vdash P_1 \ \mathbf{or}\ P_2 \Rightarrow \Gamma'''}$$

Finally, canonicalization must account for the new semantics of **or**. In order to permit replacement of variables by their values, we introduce a new compile-time-only ternary conditional operator **?:** for each variable bound on both sides of the predicate. The first argument is the predicate expression on the left-hand side of the **or** expression; the second and third arguments are the variable's values on each side of the **or**.

Canonicalizing this new **?:** expression requires ordering the tests canonically; any ordering will do. This may necessitate duplication of some expressions, such as transforming $b \mathbin{?} e_1 : (a \mathbin{?} e_2 : e_3)$ into $a \mathbin{?} (b \mathbin{?} e_1 : e_2) : (b \mathbin{?} e_1 : e_3)$ so that those two expressions are not considered distinct. With these two modifications, the tautology test is once again sound and complete.