Practical Predicate Dispatch

Todd Millstein
Computer Science Department
University of California, Los Angeles
todd@cs.ucla.edu

ABSTRACT

Predicate dispatch is an object-oriented (OO) language mechanism for determining the method implementation to be invoked upon a message send. With predicate dispatch, each method implementation includes a predicate guard specifying the conditions under which the method should be invoked, and logical implication of predicates determines the method overriding relation. Predicate dispatch naturally unifies and generalizes several common forms of dynamic dispatch, including traditional OO dispatch, multimethod dispatch, and functional-style pattern matching. Unfortunately, prior languages supporting predicate dispatch have had several deficiencies that limit its utility in practice.

We introduce JPred, a backward-compatible extension to Java supporting predicate dispatch. While prior languages with predicate dispatch have been extensions to toy or non-mainstream languages, we show how predicate dispatch can be naturally added to a traditional OO language. While prior languages with predicate dispatch have required the whole program to be available for typechecking and compilation, JPred retains Java's modular typechecking and compilation strategies. While prior languages with predicate dispatch have included special-purpose algorithms for reasoning about predicates, JPred employs general-purpose, off-the-shelf decision procedures. As a result, JPred's type system is more flexible, allowing several useful programming idioms that are spuriously rejected by those other languages. After describing the JPred language and type system, we present a case study illustrating the utility of JPred in a real-world application, including its use in the detection of several errors.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features—*classes and objects, inheritance, procedures, functions, and subroutines*

General Terms

Design, Languages

OOPSLA'04, Oct. 24-28, 2004, Vancouver, British Columbia, Canada.

Keywords

predicate dispatch, dynamic dispatch, modular typechecking

1. INTRODUCTION

Many programming languages offer a form of *dynamic dispatch*, a declarative mechanism for determining the code to be executed upon a function invocation. In this style, a function consists of a set of implementations, each with a *guard* specifying the conditions under which that implementation should be executed. When a function is invoked, all the implementations that are *applicable*, meaning that their guards are satisfied, are considered. Of the applicable implementations, the one that *overrides* all other implementations is selected to be executed.

For example, a method m in mainstream object-oriented (OO) languages like Java [3, 28] has an implicit guard specifying that the runtime class of the receiver argument must be a subclass of m's enclosing class. A method m_1 overrides another method m_2 if the enclosing class of m_1 is a subclass of the enclosing class of m_2. Multimethod dispatch, as found in languages like Cecil [11, 13] and MultiJava [16], generalizes the implicit OO guards to support runtime class tests on any subset of a method's arguments, and the overriding relation is likewise generalized to all arguments. As another example, pattern matching in functional languages like ML [40] allows guards to test the datatype constructor tags of arguments and to recursively test the substructure of arguments. In that setting, the textual ordering of function implementations determines the overriding relation.

Dynamic dispatch offers a number of important advantages over manual dispatch using `if` statements. First, dynamic dispatch allows the guards on each implementation to be declaratively specified, and the "best" implementation is automatically selected for a given invocation. Second, in the presence of OO-style inheritance, dynamic dispatch makes functions *extensible*: a function can be extended simply by writing additional implementations that override existing ones or handle new scenarios, without modifying any existing code. Finally, dynamic dispatch supports better static typechecking than manual dispatch using `if` statements. Dynamic dispatch alleviates the need for explicit runtime type casts, which subvert the static type system. Static typechecking for dynamic dispatch additionally ensures that method lookup cannot fail: there can never be dynamic *message-not-understood* errors (which occur when no methods are applicable to an invocation) or *message-ambiguous* errors (which occur when multiple methods are applicable to an invocation, but no unique applicable method overrides all others).

In 1998, Ernst *et al.* introduced the concept of *predicate dispatch* [20]. With predicate dispatch, a method implementation may specify an arbitrary predicate as a guard. A method m_1 overrides

another method m_2 if m_1's predicate logically implies m_2's predicate. Ernst *et al.* provide a number of examples illustrating how predicate dispatch unifies and generalizes several existing language concepts, including ordinary OO dynamic dispatch, multimethod dispatch, and functional-style pattern matching. They also formally define predicate evaluation and provide a static type system that ensures that method lookup cannot fail. Finally, Ernst *et al.* define a conservative algorithm for testing validity of predicates, which is necessary both for computing the method overriding relation and for static typechecking.

Despite this strong initial work, and despite additional work on the topic [14, 54, 47], to date predicate dispatch has had several deficiencies that limit its utility in practice. First, implementations of predicate dispatch have all been in the context of toy or non-mainstream languages, and none of these implementations has included static typechecking. Second, there has been no progress on static typechecking for predicate dispatch since the original work, and the type system described there is *global*, requiring access to the entire program before typechecking can be performed. This makes it difficult to ensure basic well-formedness properties of individual classes, and it clashes with the modular typechecking style of mainstream OO languages. Third, the existing static type system for predicate dispatch is overly conservative, ruling out many desirable uses of predicate dispatch. For example, that type system cannot determine that the predicates $x > 0$ and $x \leq 0$, where x is an integer argument to a function, are exhaustive and mutually exclusive. Therefore, the type system will reject a function consisting of two implementations with these guards as potentially containing both exhaustiveness and ambiguity errors. Finally, little evidence has been presented to illustrate the utility of predicate dispatch in real-world applications.

This paper remedies these deficiencies. We present JPred, a backward-compatible extension to Java supporting predicate dispatch. Our contributions are as follows:

- We illustrate through the design of JPred how predicate dispatch can be practically incorporated into a traditional OO language. The extension is small syntactically and yet makes a variety of programming idioms easier to express and validate.
- We describe a static type system for JPred that naturally respects Java's modular typechecking strategy: each compilation unit (typically a single class) can be safely typechecked in isolation, given only information about the classes and interfaces on which it explicitly depends. We achieve modular typechecking by adapting and generalizing our prior work on modular typechecking of multimethods [37].
- We describe how to use off-the-shelf decision procedures to determine relationships among predicates. We use decision procedures both to compute the method overriding relation, which affects the semantics of dynamic dispatch, and to ensure exhaustiveness and unambiguity of functions, which is part of static typechecking. The use of decision procedures provides precise reasoning about the predicates in JPred's predicate language. This contrasts with the specialized and overly conservative algorithms for reasoning about predicates that are used in previous languages containing predicate dispatch.

  Our implementation uses CVC Lite [17], a successor to the Cooperating Validity Checker [53]. CVC Lite contains decision procedures for several decidable theories, including propositional logic, rational linear arithmetic, and the theory of equality. CVC Lite is sound and complete for validity queries over JPred's predicate language, so our language and type system remain well defined and predictable.
- We have implemented JPred as an extension in the Polyglot extensible compiler framework for Java [44]. In addition to the modular typechecking strategy, we have implemented a simple modular compilation strategy that compiles JPred source to regular Java source, which can be compiled with a standard Java compiler and executed on a standard Java virtual machine. In this way, JPred source and bytecode files interoperate seamlessly with Java source and bytecode files, including precompiled Java libraries.
- To demonstrate the utility of JPred in practice, we have undertaken a realistic case study using the language. We have rewritten a Java implementation of a discovery service that is part of the *one.world* platform for pervasive computing [29] to use JPred. We illustrate and quantify the advantages that JPred provides, including its use in the detection of several errors.

Section 2 introduces JPred and illustrates its expressiveness by example. Section 3 discusses our modular static type system for JPred. Section 4 describes how we use off-the-shelf decision procedures to reason about relationships among predicates. Section 5 overviews our JPred implementation, including the modular compilation strategy. Section 6 describes the case study illustrating JPred's effectiveness. Section 7 discusses related work, and section 8 concludes.

```
pred  ::= lit | tgt
        | Identifier as tgt | [Identifier as] tgt@ClassType
        | uop pred | pred bop pred
lit   ::= null | IntegerLiteral | BooleanLiteral
tgt   ::= this | Identifier | tgt.Identifier
uop   ::= ! | -
bop   ::= && | || | == | != | < | <= | > | >= | + | - | *
```

Figure 1: The abstract syntax of predicate expressions in JPred. Brackets enclose optional pieces of syntax. Nonterminals *Identifier*, *ClassType*, *IntegerLiteral*, and *BooleanLiteral* are defined as in the Java Language Specification [28].

2. JPRED BY EXAMPLE

In this section we overview the JPred language, illustrating its benefits to programmers via a number of examples. The section ends with a precise description of the semantics of method invocation in JPred.

JPred augments the Java language by allowing each method declaration to optionally include a clause of the form `when` *pred*, just before the optional `throws` clause. The *predicate expression pred* is a boolean expression specifying the conditions under which the method may be invoked. The abstract syntax of predicate expressions is given in figure 1. Predicate expressions may include literals, references to formals and fields in scope, identifier binding, dynamic dispatch on classes, and a host of boolean, relational, and arithmetic operations. We call a method containing a `when` clause a *predicate method*.

```
class FileEditor {
  void handle(Event e) {
    if (e instanceof Open) {
      Open o = (Open) e;
      ... // open a file
    } else if (e instanceof SaveAs) {
      SaveAs s = (SaveAs) e;
      ... // save to a new file
    } else if (e instanceof Save) {
      Save s = (Save) e;
      ... // save the file
    } else {
      ... // handle unexpected events
    }
  }
}
```

Figure 2: A file editor implemented in Java.

2.1 Multimethod Dispatch

2.1.1 Event-Based Systems

Figure 2 illustrates an event-based implementation of a file editor in Java. The `handle` operation is invoked when an event is triggered by the user's action. The passed event is handled differently according to its runtime class. This implementation style for event-based systems allows multiple clients to handle posted events in different ways within an application. It also allows each client to handle a different subset of posted events. Finally, the style allows new events to be added to the system without having to modify all existing clients.

However, this style also has a number of disadvantages. First, the programmer has the burden of manually performing event dispatch, and the cases of the monolithic `if` statement must be ordered such that the right code will be executed for each scenario. For example, assuming that `SaveAs` is a subclass of `Save`, the second and third cases in figure 2 must appear in that order, or else the handler for `SaveAs` will never be invoked. Second, the monolithic style makes the event handlers difficult to reuse and extend by subclasses. For example, a subclass cannot easily choose to inherit some of `FileEditor`'s handlers, override others, and add new handlers for other events. Third, the heavy use of runtime type tests and casts provides the potential for dynamic cast failures. Finally, there is no static checking to ensure that all possible events are handled and that no handlers are redundant. For example, the `handle` method in figure 2 would still typecheck successfully if the `else` case were removed, even though that could cause errors to occur dynamically.

Figure 3 shows how the file editor can be equivalently implemented in JPred. The first three methods are *multimethods* [8], using JPred's *specializer expression* to dynamically dispatch on their arguments in addition to the receiver. Similar to multimethod notation in Cecil [11, 13], the predicate `e@Open` declares the *specialized type* (or *specializer*) of the *target* `e` to be `Open`: the first method in the figure is only applicable to an invocation of `handle` if the runtime class of the actual argument is a subclass of `Open`. When typechecking the body of the first `handle` method, the formal parameter `e` is considered to have type `Open`, thereby allowing the body to access fields and methods that are specific to the `Open` subclass of `Event`.

JPred's semantics differs from Java's static overloading mechanism, which uses the *static* type of an actual argument expression to *statically* determine which methods are applicable. For example, a Java method of the form

```
void handle(Open e) { ... // open a file }
```

will never be executed from a `handle` call site whose actual argument expression has static type `Event`, even if at runtime the actual argument is an instance of `Open`.

```
class FileEditor {
  void handle(Event e) when e@Open
    { ... // open a file }
  void handle(Event e) when e@SaveAs
    { ... // save to a new file }
  void handle(Event e) when e@Save
    { ... // save the file }
  void handle(Event e)
    { ... // handle unexpected events }
}
```

Figure 3: The file editor implemented in JPred.

```
class PrintingEditor extends FileEditor {
  void handle(Event e) when e@Open
    { ... // a better way to open files }
  void handle(Event e) when e@Print
    { ... // print the file }
}
```

Figure 4: An extension to the file editor.

The method overriding relation in JPred is determined by predicate implication; the textual order of methods is irrelevant. The `@` predicate corresponds to Java's `instanceof` expression and has the same semantics. The last `handle` method in figure 3 implicitly has the predicate `true`. Therefore, the first three methods each override the last one (since every predicate logically implies `true`), and the second method overrides the third (since an instance of `SaveAs` can always be viewed as an instance of `Save`). For example, if an invocation of `handle` dynamically has a `SaveAs` instance as the argument, then the last three methods in the figure are applicable, since their guards evaluate to true, and the second method is invoked because it overrides the third and fourth methods. While Java's `instanceof` expression can be used to test against both classes and interfaces, JPred's specializer expression requires the given type to be a class. As mentioned in section 3, this requirement allows us to preserve modular static typechecking.

The JPred implementation of the editor resolves the problems of the Java implementation in figure 2. Each conceptual handler is now encapsulated in its own method, and its guard declaratively states the conditions under which that handler should be invoked. JPred's dispatch semantics naturally matches programmer intent: the handlers can appear in any order, and JPred ensures that the right handler is invoked for each scenario. Further, the code is now statically type safe: there is no potential for dynamic cast failures, and as described in section 3, the JPred typechecker ensures that the handlers cover all possible scenarios and are not ambiguous with one another.

Finally, unlike the original implementation, the JPred implementation of `FileEditor` is easily extensible, allowing for deep hierarchies of event handlers that share code in flexible ways. Predicate methods have the same properties as regular methods, and

```
class TypeCheck {
  Type typeCheck(TypeEnv te, Node@If n) { ... }
  Type typeCheck(TypeEnv te, Node@While n) { ... }
  ...
}
```

Figure 5: Noninvasive visitors in JPred.

hence they are naturally inherited by subclasses. For example, an extended version of the editor is shown in figure 4. This editor provides a more optimized implementation of file opening and additionally provides printing functionality. JPred dispatches an invocation to one of the methods in `FileEditor` whenever no method in `PrintingEditor` is applicable. For example, if a `PrintingEditor` instance is sent a `Save` instance, the third method in figure 3 will automatically be invoked.

In practice, event handlers can be significantly more complicated than the example shown in figure 2. For example, a handler may test its current state in addition to the runtime type of the event, in order to determine how to handle the event. JPred's advantages over Java for implementing event-based systems increase as handlers become more complex. Our case study in section 6 illustrates JPred's usage in a real-world event-based system to create reliable and extensible event handlers.

As a syntactic sugar, JPred supports MultiJava-style syntax for specializer expressions [16], so the first `handle` method in figure 3 can be equivalently written as follows:

```
void handle(Event@Open e) { ... }
```

MultiJava-style specializers get desugared into a conjunction of JPred specializer expressions, which are conjoined to the front of any explicit predicate expression for the method.

2.1.2 Noninvasive Visitors

A well-known limitation of traditional OO languages is the inability to easily add new operations to existing class hierarchies. Multimethod dispatch provides a partial solution to this problem [37]. For example, figure 5 illustrates how multimethod dispatch is used to add a new typechecking pass to a hypothetical compiler. The compiler contains a class hierarchy to represent abstract syntax tree (AST) nodes, with base class `Node`. The methods in the `TypeCheck` class dynamically dispatch on different subclasses of `Node`, in order to provide functionality for typechecking the various constructs in the language being compiled.

Adding new operations to existing classes via multimethod dispatch has several advantages over use of the visitor design pattern [25], which is the standard solution in traditional OO languages. First, the visitor pattern requires the original implementer of the `Node` class and subclasses to plan ahead for visitors by including appropriate `accept` methods. This is necessary so that nodes can be dynamically dispatched upon via a *double dispatch* [32] protocol. In contrast, a JPred visitor is completely noninvasive, requiring no special-purpose "hooks" in the original nodes. Second, the visitor pattern requires all external operations to have the same argument and result types. This often requires argument and result passing to be unnaturally simulated via fields. In contrast, each JPred visitor operation can naturally have its own argument and result types, as shown in figure 5. Finally, the visitor pattern requires each visitor class to have one method per `Node` subclass, making it difficult for a node to inherit the behavior of its superclass. In contrast, a JPred visitor naturally supports inheritance among the nodes.

```
abstract class TreeNode {
  abstract TreeNode left();
  abstract TreeNode right();
  ...
}
class DataNode extends TreeNode {
  int data;
  TreeNode left;
  TreeNode right;
  ...
}
class EmptyNode extends TreeNode { ... }
```

Figure 6: A class hierarchy for binary search trees.

```
class TreeIsomorphism {
  boolean isomorphic(TreeNode@EmptyNode t1,
                     TreeNode@EmptyNode t2)
    { return true; }
  boolean isomorphic(TreeNode t1, TreeNode t2)
    when t1@EmptyNode || t2@EmptyNode
    { return false; }
  boolean isomorphic(TreeNode t1, TreeNode t2) {
    return isomorphic(t1.left(), t2.left()) &&
           isomorphic(t1.right(), t2.right());
  }
}
```

Figure 7: Disjunction in JPred predicate expressions.

2.1.3 Generalized Multimethods

While a traditional multimethod is expressed in JPred as a predicate consisting of a conjunction of specializer expressions on formals, JPred also allows arbitrary disjunctions and negations. An example of disjunction is shown in figure 7. The code operates over the class hierarchy in figure 6: `TreeNode` is the base class for binary search tree nodes, `DataNode` represents a node in the tree, and `EmptyNode` is used as a sentinel when a node lacks a left or right child (or both). The `TreeIsomorphism` class in figure 7 determines whether two binary trees (represented by their root nodes) are isomorphic. The first two methods in the figure handle scenarios when at least one of the two given tree nodes is empty. By the semantics of predicate implication, the first `isomorphic` method overrides the second one as desired.

The presence of disjunction and negation in predicates raises the issue of when to allow specialized types from a method's predicate to be used when typechecking the method's body. For example, it would be unsafe to allow `t1` to be considered to have static type `EmptyNode` when typechecking the body of the second `isomorphic` method, because that method can be invoked in a scenario where `t1` is not an instance of `EmptyNode`. In JPred we take a simple approach to handling this issue: specialized types may never "escape" from underneath disjunction and negation. Therefore, the specialized types for `t1` and `t2` are not used when typechecking the body of the second `isomorphic` method, while `t1` and `t2` may safely be considered to have static type `EmptyNode` when typechecking the body of the first method. It is possible to relax JPred's requirement, for example by allowing a specialized type for a formal that appears in both sides of a disjunction to be used when typechecking the method body [20]. However, the current rule handles the common case and is simple to understand.

```
class TypeCheck {
  ...
  Type typeCheck(TypeEnv te, Node@BinaryExpr n)
    when n.operator@Plus || n.operator@Minus
    { ... // check that the arguments are integers }
  Type typeCheck(TypeEnv te, Node@BinaryExpr n)
    when n.operator@Concat
    { ... // check that the arguments are strings }
}
```

Figure 8: Field dispatch in JPred.

```
class DataNode extends TreeNode {
  ...
  int getMin() when this.left@EmptyNode
    { return this.data; }
  int getMin() { return this.left.getMin(); }
}
```

Figure 9: Another example of field dispatch.

2.2 Field Dispatch

JPred supports dispatch on the substructure of a method's arguments, as found in functional-style pattern matching. This idiom is expressed through predicates on fields. Any field in scope within a method may be dispatched upon in the method's predicate, including fields of the receiver argument, visible fields of the other arguments, visible static fields of other classes, and fields of fields (recursively).

For example, consider the typechecking visitor in figure 5, and suppose a `BinaryExpr` subclass of `Node` represents invocations of a binary operator. It is necessary to know which binary operator is invoked in order to decide how to typecheck the invocation, and field dispatch provides a natural and declarative solution, as shown in figure 8. The example also illustrates another use of disjunction in predicates.

As another example, figure 9 uses field dispatch to find the minimum element of a binary search tree, in the context of the hierarchy in figure 6. The code mirrors the way such functionality would be naturally written in a language with pattern matching, like ML. As usual, `this.left` can equivalently be written as `left` in the predicate expression.

Unlike specialized types for formals, specialized types for fields are never used when typechecking the associated method body. For example, `this.left` is still considered to have static type `TreeNode` when typechecking the body of the first `getMin` method in figure 9, even though the method can only be invoked when `this.left` is an instance of `EmptyNode`. The `unsound` method in figure 10 illustrates why this rule is necessary. Since the static type of `n.left` is `TreeNode`, the first statement in the body of `unsound` typechecks. If the static type of `this.left` is narrowed to its specialized type `DataNode`, then the `return` statement also typechecks. However, the `return` statement will dynamically attempt to access the `data` field of an `EmptyNode` instance on an invocation `dn.unsound(dn)`, where `dn` has static type `DataNode`.

In the above example, an invocation `dn.unsound(dn)` causes `this.left` and `n.left` to be *aliases*: they have the same *l-value*. The unsoundness occurs because the two field expressions are mutable but are considered to have different types. By forcing a field expression to retain its original static type, JPred ensures type soundness regardless of how the field is modified through aliases. In

```
class DataNode extends TreeNode {
  ...
  int unsound(DataNode n) when this.left@DataNode {
    n.left = new EmptyNode();
    return this.left.data;
  }
}
```

Figure 10: Narrowing the static type of a field is unsound.

JPred, the code in figure 10 is rejected because `this.left.data` fails to typecheck — `this.left` has static type `TreeNode` and hence does not have a `data` field.

Formal parameters, unlike field expressions, are guaranteed to be unaliased: each formal parameter is given a fresh memory location. It is for this reason that the types of formal parameters may be safely narrowed to their specialized types when typechecking a method body, even in the presence of mutation. This observation provides a mechanism for safely narrowing the types of fields that have specialized types as well. JPred allows a specializer expression to bind a new identifier to the specialized target's value, and this identifier may be referenced in the associated method body. We refer to the new identifier as a *specialized name*. Specialized names are fresh local variables and hence are guaranteed to be unaliased. Therefore, it is sound to narrow the type of a specialized name to the specialized type of its target. For example, the following variant of the method in figure 10 is allowed by the JPred typechecker and is perfectly sound:

```
class DataNode extends TreeNode {
  ...
  int sound(DataNode n)
    when l as this.left@DataNode {
    n.left = new EmptyNode();
    return l.data;
  }
}
```

In the example, the method body is able to access the `data` field of `this.left`'s specialized name `l`, because `l` is considered to have static type `DataNode`. While an invocation `dn.unsound(dn)` still causes `this.left` and `n.left` to be aliases, `l` is not aliased to either of these field expressions. Therefore, assigning to `n.left` does not modify the value pointed to by `l`. The use of specialized names to soundly narrow the type of a target in JPred is reminiscent of the `focus` construct in Vault [21] and the `restrict` construct in CQUAL [23], which allow a potentially aliased location to be temporarily treated as unaliased. As with specialized types, we do not allow new identifiers bound in predicates to escape from underneath disjunction and negation.

JPred also allows a target to be bound to a new identifier using the *Identifier* `as` *tgt* syntax, without providing a specializer for the target. In that case, the new identifier acts simply as a convenient shorthand for use in the method's body. For the purposes of determining predicate implication, such a predicate is considered to be the predicate `true`, since it always succeeds (modulo null dereferences, which JPred, like Java, does not statically prevent).

2.3 Equality

Functional languages like ML allow formals and (the analogue of) fields to be tested against constants via pattern matching. JPred can express this idiom via equality testing against literals and other

```
class FileProtocol {
  static final int WANT_OPEN = 0;
  static final int WANT_CLOSE = 1;
  int state = WANT_OPEN;
  void check(Event@Open o) when state==WANT_OPEN
    { state = WANT_CLOSE; }
  void check(Event@Open o) {
    throw new FileException("Error opening file!");
  }
  void check(Event@Close c) when state==WANT_CLOSE
    { state = WANT_OPEN; }
  void check(Event@Close c) {
    throw new FileException("Error closing file!");
  }
  void check(Event e)
    { // no state change for other events }
}
```

Figure 11: A finite-state machine in JPred.

```
class ExtendedFileProtocol extends FileProtocol {
  static final int WANT_SAVE = 2;
  void check(Event@Modify m) when state==WANT_CLOSE
    { state = WANT_SAVE; }
  void check(Event@Save s) when state==WANT_SAVE
    { state = WANT_CLOSE; }
}
```

Figure 12: Extending a finite-state machine in JPred.

compile-time constant expressions. For example, `FileProtocol` in figure 11 implements a finite-state machine (FSM) that checks that users of the file editor in figure 2 never attempt two opens or two closes in a row. Typical for the implementation of FSMs in Java, the states are represented by compile-time constant fields, `WANT_OPEN` and `WANT_CLOSE`. JPred allows each transition of the FSM to be encapsulated in its own method, and the equality predicate is used to declaratively test the current state.

Unlike a corresponding implementation with functional-style pattern matching, the FSM in figure 11 is easily extensible. For example, figure 12 extends the FSM to additionally check that a modified file is saved before it is closed. One new state and two new transitions are added to the FSM.

As in Java, the `==` operator may also be used to compare objects for reference equality. (JPred disallows method invocations in predicate expressions, so equality testing via `Object`'s `equals` method is not supported.) For example, we can provide special-purpose behavior for null values. Consider the `handle` functionality in figure 3. As currently written, if `handle` is passed a null event, the only applicable method will be the last one in the figure. (Recall that the `@` predicate has the same semantics as Java's `instanceof` expression. Therefore, `null@C` is false for every class `C`.) For safety, that method's body should test whether `e` is null before attempting to access one of its fields or methods. As a declarative alternative in JPred, we can provide a separate method to handle the erroneous situation when `e` is null, which overrides the last `handle` method:

```
void handle(Event e) when e==null { ... }
```

JPred's equality predicate is more general than its analogue in functional-style pattern matching, since JPred allows targets to be

```
class DataNode extends TreeNode {
  ...
  boolean equals(Object o) when o==this
    { return true; }
  boolean equals(Object o)
    when o@DataNode && data==o.data {
    return left.equals(o.left) &&
           right.equals(o.right);
  }
}
```

Figure 13: Another example of equality predicates in JPred.

```
class DataNode extends TreeNode {
  ...
  boolean contains(int elem) when elem == data
    { return true; }
  boolean contains(int elem) when elem < data
    { return left.contains(elem); }
  boolean contains(int elem) when elem > data {
    { return right.contains(elem); }
}
```

Figure 14: Linear inequalities in JPred.

compared against one another. An example is shown in figure 13, where the `equals` method inherited from `Object` is overridden. The first method's predicate shows that JPred subsumes *alias dispatch* [34, 4], in which procedure implementations can be specialized to particular alias scenarios of their arguments. The second method's predicate tests equality of the fields of arguments to determine applicability. It also illustrates that specialized types and identifiers that escape to the method body may also be used later in the predicate expression: the `data` field of `o` may only be accessed because the type of `o` has been narrowed to `DataNode`. As shown in the example, JPred methods may override any existing methods, including those in the Java standard library.[1] If neither method in the figure is applicable to some invocation, then the default `equals` method in `Object` will automatically be selected for execution (assuming `TreeNode` does not contain an overriding `equals` method).

2.4 Linear Arithmetic

JPred supports arithmetic inequalities in predicate expressions, via the various relational and arithmetic operators shown in figure 1. All arithmetic expressions in a predicate are required to be *linear*. The JPred typechecker enforces this requirement by checking that, for every predicate expression of the form $pred_1 * pred_2$, at least one of the two operands is a compile-time constant expression as defined by Java [28]. Forcing arithmetic to be linear ensures that testing relationships among predicates, such as logical implication, remains decidable.

Figure 14 illustrates a simple example of linear inequalities in JPred. The `contains` operation checks whether a given data element is in a binary search tree. The figure shows the implementation of `contains` for `DataNode`; the implementation for `EmptyNode` is the base case and simply returns false. By predicate implication, none of the methods in the figure overrides any of the others.

As another example, consider an `nth` method on tree nodes,

[1] Technically, the methods in figure 13 should be declared `public`, in order to properly override `Object`'s `equals` method, but we have elided such modifiers for readability.

```
abstract class TreeNode {
  ...
  abstract int nth(int n);
  int nth(int n) when n < 0 {
    throw new TreeException(
      "nth invoked with a negative number!");
  }
}
```

Figure 15: Partially abstract methods in JPred.

which takes an integer argument `n` and returns the `n`th smallest element (counting from zero) in the tree. It may be desirable to make the `nth` method in the `TreeNode` class abstract, thereby forcing each subclass to provide an appropriate implementation. At the same time, it is likely that all subclasses will act identically when `nth` is passed a negative integer as an argument. Therefore, it would be nice to write the code to handle this erroneous scenario once, allowing all subclasses to inherit that functionality. In essence, we would like to make `TreeNode`'s `nth` method *partially abstract*.

Figure 15 shows how inheritance of predicate methods in JPred naturally allows operations to be declared partially abstract. The first `nth` method in `TreeNode` is declared abstract, but it is partially overridden by the second method, which handles the error case. Subclasses of `TreeNode` inherit the second method, but they are still obligated to provide implementations of `nth` that handle situations when the integer argument is nonnegative; the static type system described in section 3 enforces this obligation.

Many other operations besides `nth` naturally have error scenarios and hence would also benefit from being partially abstract. Partially abstract operations are particularly useful for large class hierarchies. For example, `TreeNode` could represent a base class for many different kinds of binary trees (binary search trees, heaps, etc.). Although each binary tree will have a different implementation of `nth`, they can all share code to handle the error case, which is nicely modularized. In contrast, Java and its type system force the programmer either to make `TreeNode`'s `nth` method (fully) abstract or to implement it for all possible integer values.

2.5 Method Invocation Semantics

We end this section by describing more precisely the semantics of method invocation in JPred. To simplify the discussion, we assume that all methods have a `when` clause and that MultiJava-style specializers have been desugared. A method without a `when` clause is equivalent to one with the clause `when true`.

Consider a message send of the form $e_1.m(e_2,\ldots,e_n)$ appearing in some JPred program. At compile time, static overload resolution is performed exactly as in Java, based on the name *m* and the static types of $e_1,\ldots,e_n$. This has the effect of determining which *generic function* [8, 41] — a collection of methods of the same name, number of arguments, and static argument and result types — will be invoked dynamically.

At runtime, each expression e_i is evaluated to produce a value v_i and the *unique most-specific applicable method* belonging to the statically determined generic function is invoked. A method is *applicable* if its associated predicate expression evaluates to true in the context of the given actual arguments $v_1,\ldots,v_n$. A method is the unique most-specific applicable method if it is the unique applicable method that *overrides* all other applicable methods. Finally, one method m_1 overrides another method m_2 if either of the following holds:

```
class C {
  private Object f;
  void m() when f@String { ... }
  ...
}
class D extends C {
  Object g;
  void m() when g@String { ... }
  ...
}
```

Figure 16: A problem with the symmetric approach to method lookup.

- Method m_1's receiver class is a strict subclass of m_2's receiver class.
- Methods m_1 and m_2 are declared in the same class, and m_1's predicate expression logically implies m_2's predicate expression. We use off-the-shelf decision procedures, which are discussed in section 4, to test predicate implication.

For example, consider the invocation `treeIso.isomorphism(en, dn)` in the context of the class in figure 7, where `treeIso`, `en`, and `dn` have runtime types `TreeIsomorphism`, `EmptyNode`, and `DataNode`, respectively. The second and third methods in the figure are applicable, and the second method is the unique most-specific applicable one.

If there are no applicable methods for a message send, a message-not-understood error occurs. If there is at least one applicable method but no unique most-specific applicable method, a message-ambiguous error occurs. The modular static type system in section 3 ensures that these kinds of errors cannot occur.

JPred's method-lookup semantics can be viewed as a generalization of the *encapsulated* style of multimethod dispatch [10]. In this style, dispatch consists of two phases. In the first phase, ordinary OO-style dispatch finds the receiver argument's class. In the second phase, the unique most-specific applicable method in the receiver class is selected, recursively considering methods in the superclass if no methods in the receiver are applicable.

Other multimethod semantics could instead be generalized in JPred without affecting our results. For example, we could generalize the *symmetric* multimethod semantics, in which the receiver argument is not treated specially. This semantics is used in multimethod languages such as Cecil and MultiJava. In the symmetric approach to JPred dispatch, a method m_1 would be considered to override another method m_2 only if m_1's receiver class is a (reflexive, transitive) subclass of m_2 *and* m_1's predicate expression logically implies m_2's predicate expression.

We chose the encapsulated style in JPred for several reasons. First, the encapsulated style is arguably quite natural in a language like Java, which is already heavily receiver-centric. Further, the encapsulated style reduces the dependence of classes on their superclasses: a class's methods cannot be ambiguous with any methods in superclasses. Finally, receiver-based encapsulation can sometimes make the symmetric semantics impossible to satisfy, as shown in figure 16. Under JPred's invocation semantics, `D`'s `m` method overrides `C`'s `m` method. In contrast, under the symmetric semantics, neither of the two methods in the figure is considered to override the other: a message-ambiguous error will occur if `m` is ever invoked on a `D` instance whose `f` and `g` fields are both instances of `String`.

Therefore, under the symmetric semantics a static typechecker must reject the program in figure 16. If `f` were accessible from `D`, then the implementer of `D` could resolve the ambiguity and allow the program to typecheck by adding a new method as follows:

```
void m() when f@String && g@String { ... }
```

Since `f` is private to `C`, however, there is no way for the implementer of `D` to resolve the ambiguity. Indeed, under the symmetric semantics, *every* `m` method in class `D` (except methods whose predicates are logically false) is ambiguous with `C`'s `m` method.

3. STATIC TYPECHECKING

This section describes our extensions to Java's static type system to support predicate methods. A key feature is the retention of Java's *modular* typechecking approach, whereby each compilation unit can be typechecked separately, given type information about the other compilation units on which it depends. Others have formalized Java's notion of modular typechecking and compilation [19, 2].

3.1 Typechecking Message Sends

Message sends are typechecked in JPred exactly as in Java; no modifications are required. As mentioned in section 2.5, Java's static overload resolution is performed to determine which generic function a message send invokes, based on the message name, number of arguments, and static types of the argument expressions. As usual, the result type of the generic function is then used as the type of the entire message send expression.

3.2 Typechecking Method Declarations

Typechecking for method declarations is augmented to reason about `when` clauses. First we describe the local checks performed on each predicate method in isolation. Then we describe the checks ensuring that a generic function's methods are *exhaustive*, so that message-not-understood errors cannot happen at runtime. Finally, we describe the checks ensuring that a generic function's methods are *unambiguous*, so that message-ambiguous errors cannot happen at runtime.

3.2.1 Local Checks

Local checks on a predicate method are largely straightforward. The main requirement is that the method's associated predicate expression typechecks successfully and has the type `boolean`. The predicate expressions (see figure 1) that are also legal Java expressions — literals, targets, unary predicate expressions, and binary predicate expressions — are typechecked exactly as they are in Java. Arithmetic predicate expressions are additionally checked to be linear, as described in section 2.4. Specializer expressions of the form *tgt*`@`*ClassType* are typechecked like Java's `instanceof` expressions, with the additional constraint that the specialized type be a class that is a strict subtype of the target's static type. Specialized types may not be interfaces because of challenges for modular exhaustiveness and ambiguity checking of multimethod dispatch in the presence of multiple inheritance [39].

Specializer expressions of the form *Identifier* `as` *tgt*`@`*ClassType* are typechecked like regular specializer expressions, and the specialized name *Identifier* is additionally given the static type *ClassType*. An identifier binding of the form *Identifier* `as` *tgt* is typechecked by typechecking the target as in Java and additionally giving *Identifier* the static type determined for the target. It is an error if an identifier is bound more than once in a predicate. Specialized types for formals and type bindings for identifiers that can escape to the method body are used when typechecking the method body. They also propagate from left to right during the typechecking of the predicate itself.

A predicate method may not be declared `abstract`. However, concrete predicate methods are allowed in abstract classes, and they can be used to implement partially abstract methods, as shown in figure 15. Consistent with Java, a predicate method may have weaker access restrictions than overridden methods in superclasses, and it may declare a subset of the exceptions declared by overridden methods in superclasses. However, we require a predicate method to have the same modifiers and declared exceptions as all other methods belonging to the same generic function that are in the same class. It is possible to relax this rule, analogous with Java's requirements. For example, it would be sound to allow the first `check` method in figure 11 to be declared `public`, since the two methods it overrides are both implicitly package-visible.

We have decided not to allow this relaxation, since it requires the ability to statically evaluate predicate expressions in order to be useful. For example, if the first `check` method were declared `public`, an invocation of `check` from outside of `FileProtocol`'s package could only be allowed to typecheck if the JPred typechecker could statically prove that the argument event is an instance of `Open` or a subclass and the receiver's `state` field is equal to `WANT_OPEN`. Rather than forcing the type system to incorporate a conservative analysis for statically evaluating predicate expressions, our current rule gives up a bit of flexibility, keeping the type system simple yet still backward-compatible with Java's type system.

3.2.2 Exhaustiveness Checking

Exhaustiveness checking ensures that message-not-understood errors will not occur in well-typed programs: each type-correct tuple of arguments to a message has at least one applicable method. Such checking is already a part of Java's modular typechecks. For example, a static error is signaled in Java if a concrete class does not implement an inherited abstract method, because that situation could lead to a dynamic `NoSuchMethodException`, the equivalent of our message-not-understood error.

JPred naturally augments Java's class-by-class exhaustiveness checking. As in Java, for each concrete class *C* we check that *C* implements any inherited abstract methods. For example, assuming that `TreeNode`'s `contains` method is declared `abstract`, `DataNode` in figure 14 will be checked to implement `contains` for all possible scenarios. In JPred we must also check that *C* implements any inherited partially abstract methods. For example, `DataNode` will be checked to implement the partially abstract `nth` method in figure 15 for all nonnegative integer arguments. Finally, in JPred we must check that *C* fully implements any new generic functions declared in *C* (i.e., methods in *C* that have no overridden methods in superclasses). For example, `FileEditor` in figure 3 will be checked to implement the new `handle` generic function for all possible pairs of argument events.

All of these checks are performed in a uniform way. To check exhaustiveness of a generic function from a class *C*, we collect all of the concrete methods of that generic function declared in *C* and inherited from superclasses of *C*. If at least one of these methods is a regular Java method (i.e., it has no `when` clause), then exhaustiveness is assured and the check succeeds. Otherwise, all of the methods are predicate methods, and the check succeeds if the disjunction of all of the methods' predicates is logically valid.

For example, consider exhaustiveness checking of `handle` in figure 3. Since the last method has no `when` clause, the check succeeds. As another example, consider exhaustiveness checking of

`contains` for `DataNode` in figure 14. None of the declared methods is a regular Java method, but the disjunction of the methods' predicates is logically valid (since one integer is always either equal to, less than, or greater than another integer), so the check succeeds.

Our exhaustiveness checking algorithm is conservative in the face of partial program knowledge, which is critical for modular typechecking. For example, consider again exhaustiveness checking for `handle` in figure 3, and suppose that the last method were missing. In that case, our typechecker would signal a static exhaustiveness error, since the disjunction of the remaining methods' predicates is not valid. Indeed, it is possible that there exist concrete subclasses of `Event` other than `Open`, `Save`, and `SaveAs`, and these events may not be visible from `FileEditor`.[2]

Java and MultiJava share JPred's conservatism, and in fact those languages are strictly more conservative than JPred. Both Java and MultiJava always require the existence of a *default* method, which handles all possible arguments of the appropriate type, to ensure exhaustiveness. In contrast, JPred can sometimes safely ensure exhaustiveness without forcing the existence of a default method, as shown in the `contains` example above. Ernst *et al.*'s exhaustiveness checking algorithm for predicate dispatch [20] safely requires fewer default methods than JPred, but the algorithm can only be performed when the whole program is available. In the `handle` example, `Open` and `Save` must be known to be the only direct concrete subclasses of `Event` in the entire program in order to ensure exhaustiveness without requiring the last `handle` method.

3.2.3 Ambiguity Checking

Ambiguity checking ensures that message-ambiguous errors will not occur in well-typed programs: if a type-correct tuple of arguments to a message has at least one applicable method, then it has a unique most-specific applicable method. Again, such checking is already a part of Java's modular typechecks. In particular, Java signals a static error if a class contains two methods of the same name, number of arguments, and static argument types. Languages that support multiple inheritance, like C++ [52], perform additional ambiguity checks modularly.

JPred performs ambiguity checking for each class C by comparing each pair of methods declared in C that belong to the same generic function. The algorithm for checking each method pair generalizes our earlier algorithm for modular ambiguity checking in Extensible ML (EML) [38] to handle JPred's predicate expressions, which subsume EML's pattern-matching facility. Consider a pair of methods m_1 and m_2. If each method overrides the other, then the methods have the same logical predicate and hence are ambiguous. This check subsumes Java's check for duplicate methods. If one method overrides the other, but not vice versa, then one method is strictly more specific than the other, so the methods are not ambiguous.

Finally, suppose neither method overrides the other. Then m_1 and m_2 are predicate methods, with predicates $pred_1$ and $pred_2$, respectively. There are two cases to consider. If the methods are *disjoint*, meaning that they cannot be simultaneously applicable, then they are not ambiguous. The methods are disjoint if $pred_1$ and $pred_2$ are mutually exclusive: $\neg(pred_1 \wedge pred_2)$ is valid. If the methods are not disjoint, then the methods are ambiguous unless the set $\overline{m}$ of methods that override both m_1 and m_2 is a *resolving set*, meaning that at least one member of $\overline{m}$ is applicable whenever both m_1 and m_2 are applicable. The set $\overline{m}$, with associated predicates $\overline{pred}$, is a resolving set if $((pred_1 \wedge pred_2) \Rightarrow \bigvee \overline{pred})$ is valid.

Consider ambiguity checking for `check` in `FileProtocol` of figure 11. There are ten pairs of methods to consider. The first four methods each override the last method, but not vice versa, so these pairs are unambiguous. The pair consisting of the first and second methods passes the check similarly, as does the pair consisting of the third and fourth methods. Finally, each of the first two methods is disjoint from each of the third and fourth methods. Therefore, ambiguity checking for `check` in `FileProtocol` succeeds.

To illustrate resolving sets, consider an `OpenAs` subclass of `Open`, which copies a file to a new name and opens it, and suppose `FileProtocol` contained a method of the following form:

```
void check(Event@OpenAs o) { ... }
```

In that case, the JPred typechecker would signal a static error indicating that the above method is ambiguous with the first `check` method in figure 11: the methods are not disjoint and there are no methods that override both of them, so the test for a resolving set fails trivially. Indeed, the two methods will cause a dynamic message-ambiguous error if `check` is ever passed an `OpenAs` event when the receiver is in the `WANT_OPEN` state. However, the ambiguity would be resolved, and typechecking would succeed, if `FileProtocol` additionally contained a method of the following form:

```
void check(Event@OpenAs o) when state==WANT_OPEN
  { ... }
```

JPred's ambiguity checking algorithm is naturally modular: only pairs of methods declared in the same class are considered. The semantics of method invocation described in section 2.5 ensures that two methods declared in different classes cannot be ambiguous with one another. If one method's class is a strict subclass of the other method's class, then the first method overrides the second. Otherwise, neither method's class is a subclass of the other, so the methods are guaranteed to be disjoint.

JPred's modular ambiguity checking algorithm is very similar to the original ambiguity algorithm for predicate dispatch [20]. However, that algorithm is performed on all pairs of methods belonging to the same generic function in the entire program. Further, that algorithm does not check for a set of resolving methods, instead conservatively rejecting the program whenever two methods are not in an overriding relation and are not disjoint.

4. AUTOMATICALLY REASONING ABOUT PREDICATES

As described in sections 2 and 3, both the dynamic and static semantics of JPred rely on the ability to test relationships among predicate expressions. All of these tests reduce to the ability to check validity of propositional combinations of formulas expressible in JPred's predicate language. Prior languages containing predicate dispatch have used their own specialized algorithms for conservatively checking validity of predicates [20, 54, 47]. In contrast, JPred employs general-purpose off-the-shelf decision procedures, which are more flexible and precise than these specialized algorithms.

In particular, we rely on an automatic theorem prover that consists of a combination of decision procedures for various logical theories. Using an automatic theorem prover as a black box allows us to easily incorporate advances in decision procedures as they arise. For example, the search for more efficient decision procedures for propositional satisfiability is an active area of research. Using an automatic theorem prover also makes it easier to augment our language with new kinds of predicates. Rather than being forced to extend a specialized validity algorithm to handle the

[2] These events may not even have been written when `FileEditor` is typechecked and compiled.

new predicates, we have the simpler task of deciding how to appropriately represent the new predicates in the logic accepted by the prover.

In this section we describe the interface between JPred and an automatic theorem prover. First we describe CVC Lite [17], which is the automatic theorem prover that our implementation uses. Then we describe how JPred's predicate expressions are represented in CVC Lite's input language. Finally we describe the axioms we provide to CVC Lite so it can reason precisely about objects and classes.

4.1 CVC Lite

CVC Lite is an automatic theorem prover in the Nelson-Oppen style [43]. The theorem prover integrates separate decision procedures for several decidable theories, including:

- propositional logic
- rational linear arithmetic
- the theory of equality with uninterpreted function symbols
- the theory of arrays

CVC Lite also allows users to plug in decision procedures for other theories, but we have not exploited this feature.

CVC Lite's input language allows expression of quantifier-free first-order formulas over the above interpreted theories. The logic of CVC Lite's input language is decidable, so CVC Lite is sound, complete, and fully automatic.[3] In a typical usage, various formulas are provided as *axioms*, which CVC Lite assumes to be true. These user-defined axioms, along with the axioms and inference rules of the underlying theories, are then used to automatically decide whether a *query formula* is valid.

For our purposes, there is nothing special about CVC Lite. There are several automatic theorem provers of comparable expressiveness to CVC Lite, including Simplify [18], Vampyre [7], CVC [53], and Verifun [22]. Moving to one of these provers would merely require us to translate the queries and axioms we provide to CVC Lite (see the next two subsections) into the input language of the new prover.

4.2 Representing Predicate Expressions

Before translating a predicate expression into the syntax of CVC Lite's input language, we convert it to *internal form*. This conversion process canonicalizes the predicate expression so it can be properly compared to a predicate expression of another method. First, we replace the *i*th formal name with the name `arg`*i* everywhere.[4] Second, we replace any compile-time constant expressions with their constant values. Third, we convert targets to their full names, for example adding a prefix of `this` if it was left implicit. Fourth, we substitute any use of a bound identifier in the predicate expression with the identifier's associated target expression, which is itself recursively internalized. Finally, we remove identifier bindings. Ordinary identifier binding expressions are replaced by `true`, and specialized identifier bindings simply have the binding removed, leaving the specializer expression.

[3] An extension to CVC Lite allows formulas to contain explicit universal and existential quantification. The resulting logic is undecidable, so CVC Lite uses various heuristics to prove such formulas valid, remaining sound but losing completeness. JPred does not make use of this extension to CVC Lite.

[4] The actual internal-form argument names are slightly more complicated, to prevent accidental clashes with other variable names in the program. We elide the issue of name mangling throughout this section.

It is straightforward to translate predicate expressions in internal form into the syntax of CVC Lite's input language. All of our allowed unary and binary operators (see figure 1) are translated to their counterparts in CVC Lite, as are all of the literals except `null`. The literal `null` and all targets appearing in a given predicate expression are translated to themselves; they are treated as variable names by CVC Lite. For example, the target `this.data` is treated as an atomic variable name, with no relation to the target `this`. Finally, a specializer expression *tgt*`@`*ClassType* is translated as `instanceof(`*tgt*`,` *ClassType*`)`, where `instanceof` is an uninterpreted function symbol that we declare.

For example, consider the `contains` methods in `DataNode` of figure 14. During static ambiguity checking, the first and second methods are tested for disjointness by posing the following query to CVC Lite:

```
NOT(arg1 = this.data AND arg1 < this.data)
```

Here `=` is CVC Lite's analogue of Java's `==` operator. CVC Lite easily proves this formula to be valid, because of the relationship between `=` and `<`, so the methods are proven to be disjoint and hence unambiguous.

4.3 Axioms

Consider testing disjointness of the first and fourth `check` methods in figure 11. After converting their predicates to internal form, we pose the following query to CVC Lite:

```
NOT((instanceof(arg1, Open) AND this.state = 0) AND
    instanceof(arg1, Close))
```

Since `instanceof` is an uninterpreted function symbol, CVC Lite does not know anything about its semantics. Therefore, CVC Lite cannot prove that the above formula is valid, even though the two methods are in fact disjoint.

To address this issue, we provide CVC Lite with axioms about the semantics of `instanceof`. These axioms effectively mirror the relevant portion of the associated JPred program's class hierarchy in CVC Lite. We call a target a *reference target* if it has reference type. Let F be a query formula provided to CVC Lite. We declare one axiom per pair $(\{C_1, C_2\}, tgt)$, where C_1 and C_2 are distinct class names mentioned in F and *tgt* is a reference target mentioned in F. In particular:

- If C_1 is a subclass of C_2, we declare the axiom `instanceof(`*tgt*`,` C_1`) => instanceof(`*tgt*`,` C_2`)`, where `=>` is the logical implication operator in CVC Lite.
- If C_2 is a subclass of C_1, we declare the axiom `instanceof(`*tgt*`,` C_2`) => instanceof(`*tgt*`,` C_1`)`.
- If C_1 is not a subclass of C_2 and C_2 is not a subclass of C_1, we declare the axiom `NOT(instanceof(`*tgt*`,` C_1`) AND instanceof(`*tgt*`,` C_2`))`.

For the `check` example query above, we automatically produce the following axiom:

```
NOT(instanceof(arg1, Open) AND
    instanceof(arg1, Close))
```

In the presence of this axiom, CVC Lite can now prove that the above query is valid, and hence the JPred typechecker will correctly conclude that the first and fourth `check` methods in figure 11 are disjoint. Our axioms for `instanceof` are similar to the implication rules used to rule out infeasible truth assignments in Ernst *et al.*'s special-purpose algorithm for validity checking [20].

To complete the semantics of `instanceof`, we have two other kinds of axioms. First, for each class name C in a query formula F, we declare the axiom `NOT(instanceof(null, C))`. This axiom reflects the fact that JPred's specializer expression evaluates to false whenever the target is null. The axiom allows JPred to properly conclude that the `handle` method near the end of section 2.3, which tests whether the argument event is null, is disjoint from each of the first three `handle` methods in figure 3. Second, for every reference target *tgt* in F, we declare the axiom `instanceof(`*tgt*`, C) OR` *tgt* `= null`, where C is the static type of *tgt*. For the special reference target `this`, which can never be null, we leave off the second disjunct in this axiom.

Finally, we provide CVC with axioms that relate reference targets and their fields. For each set of targets $\{tgt_1, tgt_2, tgt_1.f, tgt_2.f\}$ mentioned in a query formula F, we declare the axiom

$$tgt_1 = tgt_2 \Rightarrow tgt_1.f = tgt_2.f$$

This axiom allows JPred to properly conclude that the first `equals` method in figure 13 overrides the second one.

5. IMPLEMENTATION

We have implemented JPred in the Polyglot extensible compiler framework for Java [44], declaring new abstract syntax tree (AST) nodes to represent predicate expressions and predicate methods. In this section, we describe our augmentations to Polyglot for typechecking and compiling a class containing predicate methods.

5.1 Typechecking

The local checks on predicate methods, described in section 3.2.1, are performed on each predicate method as part of Polyglot's existing typechecking pass. In a subsequent pass, we partition a class's methods by generic function: each generic function's methods, where at least one of the methods is a predicate method, are collected in a *dispatcher*; the methods belonging to the same dispatcher are called *dispatcher mates* [15]. Finally, we perform exhaustiveness and ambiguity checking on each dispatcher in a class, using the algorithms described in section 3. This checking involves sending queries to CVC Lite using the translation and axioms described in section 4. As part of ambiguity checking we compute the method overriding partial order, which is also used during code generation.

5.2 Code Generation

We generate code for JPred in two steps. First we rewrite all of the JPred-specific AST nodes into Java AST nodes. Then we use an existing pass in Polyglot to output a source-code representation of Java AST nodes. The resulting `.java` files can be compiled with a standard Java compiler and executed on a standard Java virtual machine.

Our modular compilation strategy generalizes that of MultiJava [16, 15] to handle JPred's predicate language. First, there are several modifications to each method *m* associated with a dispatcher. We modify *m* to be declared `private` and to have a unique name. If *m* has a predicate *pred*, we add a new local variable at the beginning of *m*'s body for each identifier bound in *pred* that escapes to the body. The new local variable is initialized with the identifier's corresponding target, which is first cast to the associated specialized type, if any. Static typechecking has ensured that this cast cannot fail dynamically. If a formal parameter is specialized in *pred* and that specialized type escapes to the body, we replace the formal's original static type in *m* with the specialized type. Finally, *m*'s associated `when` clause is removed.

```
private boolean isomorphic1(EmptyNode t1,
                            EmptyNode t2)
  { return true; }
private boolean isomorphic2(TreeNode t1,
                            TreeNode t2)
  { return false; }
private boolean isomorphic3(TreeNode t1,
                            TreeNode t2) {
  return isomorphic(t1.left(), t2.left()) &&
         isomorphic(t2.right(), t2.right());
}
```

Figure 17: The translation of the `isomorphic` methods in figure 7 to Java.

```
boolean isomorphic(TreeNode arg1, TreeNode arg2) {
  if (arg1 instanceof EmptyNode &&
      arg2 instanceof EmptyNode) {
    return isomorphic1((EmptyNode) arg1,
                       (EmptyNode) arg2);
  } else if (arg1 instanceof EmptyNode ||
             arg2 instanceof EmptyNode) {
      return isomorphic2(arg1, arg2);
  } else {
      return isomorphic3(arg1, arg2);
  }
}
```

Figure 18: A dispatcher method for the methods in figure 17.

For example, figure 17 illustrates the Java translation of the `isomorphic` methods from figure 7. In the first method, the static argument types have been narrowed to reflect their specialized types. In the second method, which corresponds to the original method with predicate `t1@EmptyNode || t2@EmptyNode`, the argument types are unchanged, because neither specializer escapes to the body.

To complete the translation from JPred to Java, we create a *dispatcher method* for each dispatcher *d*. The method has the same name, modifiers, and static argument and return types as the original methods associated with *d*. Therefore, compilation of clients of the generic function is unchanged. The body of the dispatcher method uses an `if` statement to test the guards of each associated method one by one, from most- to least-specific, in some total order consistent with the method overriding partial order. The first method whose guard evaluates to true is invoked. Static ambiguity checking ensures that this method is in fact the unique most-specific applicable method. If all the methods in *d* are predicate methods and there exist inherited methods belonging to the same generic function, the last branch of the dispatcher method's `if` statement uses `super` to recursively invoke the superclass's dispatcher method. Static exhaustiveness checking ensures that an applicable method will eventually be found.

Figure 18 contains the dispatcher method for the methods in figure 17. The dispatcher method is given a canonical set of formal names. Each method's predicate is tested by converting the predicate to internal form, described in section 4.2, and replacing each specializer expression with its equivalent `instanceof` expression. Although not necessary in this example, the internal form of a predicate is also augmented with casts, wherever a target is substituted for its corresponding specialized name and wherever a formal with a specialized type that can escape to the body is referenced. Simi-

```
public interface EventHandler {
  void handle(Event event);
}
```

Figure 19: The interface for event handlers in *one.world*.

larly, a formal with an escaping specialized type must be cast to the specialized type before invoking a predicate's associated method, as shown in the first branch of the `if` statement. As above, static typechecking has ensured that none of these casts can fail.

Because the original methods are now private, calls to them from the dispatcher method are statically bound and therefore do not incur the performance overhead of dynamic dispatch. A Java compiler can inline these methods in the dispatcher method, to further reduce overhead.

6. CASE STUDY

This section describes a case study that we undertook to evaluate JPred's effectiveness in a realistic setting. First we describe the Java application that we rewrote in JPred. Then we illustrate the basic technique we used to perform the translation from Java to JPred. Finally, we discuss results of the case study, including the detection of several errors.

6.1 Application

The *one.world* system is a framework for building pervasive applications in Java, designed and implemented by others [29]. Users build applications in *one.world* as collections of *components* that communicate through asynchronous events. Each component *C* imports a set of *event handlers*, to which *C* can pass events, and likewise exports a set of event handlers, to which others can pass events meant for *C*.

The *one.world* system is implemented as a class library in Java. Users write *one.world* components by subclassing from the abstract `Component` class. Event handlers in *one.world* use the same style as the event handlers in our `FileEditor` example in figure 2. In particular, *one.world* event handlers meet the simple interface shown in figure 19: an event handler provides a `handle` method, which is passed the event that occurs. A component's exported event handlers are typically implemented as inner classes. The set of handlers that a component imports is initially decided during static initialization, but it can also be modified dynamically. Having all event handlers meet the same interface facilitates such dynamic rebinding.

The *one.world* system includes a set of basic services that helps programmers build applications that meet the unique demands of pervasive computing. These services are themselves written in the component-based style described above. One such service is a *discovery service*, which allows a component to query for event handlers that satisfy a particular description; the querying component can then import the resulting event handler(s) and begin communication. A canonical example is a component that queries for a printer in the current environment, which can subsequently be sent files to be printed. The discovery service in *one.world* supports several varieties of querying and communication, which are described elsewhere [29].

6.2 Overview

In this case study, we rewrote the event handlers in the implementation of *one.world*'s discovery service to use JPred. We started with the discovery service implementation from *one.world* version 0.7.1, which is freely available for download [46]. The discovery service implementation consists of two components, `DiscoveryClient` and `DiscoveryServer`, totaling 2371 non-comment, non-blank lines of code (LOC). Together the two components include 11 event handlers as inner classes totaling 977 LOC, or 41% of the code.

Figure 20 shows the Java implementation of an event handler of average complexity in `DiscoveryClient`.[5] Handlers typically subclass from the abstract `AbstractHandler` class, which in turn implements the `EventHandler` interface. `AbstractHandler`'s `handle` method invokes an abstract helper method `handle1`, which is implemented by each concrete subclass in order to provide the subclass's event-handling functionality. A `handle1` method should return true if the associated subclass was able to successfully handle the given event and false otherwise. The implementation of `AbstractHandler` is discussed further in section 6.3.4.

Figure 21 illustrates how we implement `MainHandler` from figure 20 in JPred. All of the advantages of JPred for event-based systems, as described in section 2.1, apply to our *one.world* case study. Unlike the Java version, the JPred implementation is declarative and statically typesafe, removing a large source of potential runtime errors. The JPred implementation is also extensible, opening up the possibility of fine-grained handler reuse in *one.world*. For example, a subclass of `DiscoveryClient` could contain an inner handler that subclasses from `MainHandler` (if it were not declared `final`), inherits some of `MainHandler`'s `handle1` methods, overrides others, and handles new scenarios with additional `handle1` methods.

The JPred style of implementing event handlers is very natural. This is illustrated by the fact that the original implementers of the discovery service often manually simulated JPred-style decomposition, as shown in figure 22. To manage the complexity of event handling in `ServerManager`, the code has been divided into two layers. The `handle1` method manually dispatches on the event (and on the `event` field of a `RemoteEvent`) in order to invoke an appropriate private helper method (whose implementation is not shown), which actually handles the event. With JPred, each helper method naturally becomes one or more `handle1` methods, each declaratively specifying its dispatch constraints. The original `handle1` method is then removed, since JPred automatically dispatches to the appropriate handler. Indeed, as described in section 5.2 the JPred compiler will generate a dispatcher method that is almost identical to the `handle1` method in figure 22.

The handlers in figure 21 that only dispatch on the event's runtime type could be implemented in MultiJava. In fact, MultiJava has been successfully used to implement other event-based systems [37], and this experience led us to use an event-based system for the current case study. The handlers that dispatch on fields cannot be directly expressed in MultiJava. Instead, helper methods must be created to perform field dispatch. For example, the translation to MultiJava of the fourth and fifth methods in figure 21 is shown in figure 23. This style is tedious and forces the dispatching logic to be spread across multiple generic functions. Further, as we show in section 6.3.2 below, some predicates cannot be expressed at all in MultiJava.

Although our case study only involves rewriting the implementation of *one.world*'s discovery service, the ways in which we employ JPred are more general. JPred would provide similar benefits for other services provided by *one.world* as well as for applications written by programmers on top of *one.world*. Further, our usage of JPred would apply to the implementation of event-based sys-

[5] In examples throughout this section, we elide comments and code used for debugging.

```
final class MainHandler extends AbstractHandler {
  protected boolean handle1(Event e) {
    if (e instanceof EnvironmentEvent) {
      EnvironmentEvent ee = (EnvironmentEvent) e;
      switch(ee.type) {
        case EnvironmentEvent.ACTIVATED:
          ...
          break;
        default:
          ...
          break;
      }
      return true;
    } else if (e instanceof BindingResponse) {
      return true;
    } else if (e instanceof InputResponse) {
      InputResponse ir = (InputResponse) e;
      if (ir.tuple instanceof AnnounceEvent) {
        AnnounceEvent ae = (AnnounceEvent) ir.tuple;
        ...
      }
      return true;
    } else if (e instanceof ListenResponse) {
      ListenResponse lr = (ListenResponse) e;
      ...
      return true;
    }
    return false;
  }
}
```

Figure 20: An event handler in `DiscoveryClient`**.**

```
final class MainHandler extends AbstractHandler {
  protected boolean handle1(Event@EnvironmentEvent ee)
    when ee.type == EnvironmentEvent.ACTIVATED {
      ...
      return true;
  }
  protected boolean handle1(Event@EnvironmentEvent ee) {
      ...
      return true;
  }
  protected boolean handle1(Event@BindingResponse br) {
    return true;
  }
  protected boolean handle1(Event@InputResponse ir) {
    return true;
  }
  protected boolean handle1(Event@InputResponse ir)
    when ae as ir.tuple@AnnounceEvent {
      ...
      return true;
  }
  protected boolean handle1(Event@ListenResponse lr) {
    ...
    return true;
  }
  protected boolean handle1(Event e) {
    return false;
  }
}
```

Figure 21: The translation of the code in figure 20 to JPred.

```
final class ServerManager extends AbstractHandler {
  protected boolean handle1(Event e) {
    if (e instanceof RemoteEvent) {
      RemoteEvent re = (RemoteEvent) e;
      if (re.event instanceof BindingResponse) {
        handleBindingResponse(re.source,
                              (BindingResponse) re.event);
        return true;
      } else if (re.event instanceof LeaseEvent) {
        handleLeaseEvent(re.source,
                         (LeaseEvent) re.event);
        return true;
      } else if (re.event instanceof ExceptionalEvent) {
        ExceptionalEvent eev = (ExceptionalEvent) re.event;
        ...
      }
      return false;
    } else if (e instanceof LeaseRenew) {
      handleLeaseRenew((LeaseRenew) e);
      return true;
    } else if (e instanceof ServerCheck) {
      handleServerCheck((ServerCheck) e);
      return true;
    } else if (e instanceof EntryEvent) {
      handleEntryEvent((EntryEvent) e);
      return true;
    } else if (e instanceof ExceptionalEvent) {
      handleExceptionalEvent((ExceptionalEvent) e);
      return true;
    } else {
      return false;
    }
  }
}
```

Figure 22: Another event handler in `DiscoveryClient`**.**

```
protected boolean handle1(Event@InputResponse ir) {
  return handleIR(ir, ir.tuple);
}
protected boolean handleIR(InputResponse ir,
                           Tuple tuple)
  { return true; }
protected boolean handleIR(
  InputResponse ir, Tuple@AnnounceEvent tuple) {
  ...
  return true;
}
```

Figure 23: Simulating field dispatch in MultiJava. `Tuple` **is a superclass of** `Event` **and is the static type of the** `tuple` **field of** `InputResponse`**.**

Table 1: Quantitative results.

	Java	JPred
methods	20	87
total LOC	844	736
avg LOC	42	8.5
max LOC	181	44
`instanceofs`	80	6
casts	82	18
compile time (sec)	4.5	8.6
CVC Lite queries	N/A	452

tems other than *one.world*. The event-based style has been recommended for the implementation of many important classes of applications, from Internet services [48, 56] to sensor networks [30, 27]. Finally, section 2 illustrated other uses of JPred, for example to implement compilers and data structures.

6.3 Results

6.3.1 Quantitative Results

We can quantify several basic properties of the original and rewritten discovery service implementation. These properties are described in table 1. "Java" refers to the original implementation of the code, and "JPred" refers to the version rewritten in JPred. Twenty original methods were rewritten to use JPred's features. These include nine `handle1` methods and 11 helper methods like `handleBindingResponse`, whose usage is illustrated in figure 22. These methods have an average size of 42 LOC, with the biggest method being 181 LOC. In the JPred implementation, the elimination of manual dispatching logic causes the number of methods to roughly quadruple and the total code size to be reduced, leading to corresponding reductions in the average and maximum method sizes. The small method sizes indicate that each logical handler is quite simple. JPred allows these conceptually distinct handlers to be written as distinct methods, whose headers declaratively specify their applicability constraints and whose bodies are easy to understand.

The table shows the number of `instanceofs` and casts present in the original and rewritten methods. Almost all of the manual event dispatching is obviated by the JPred style. The six remaining `instanceofs` could be removed by introducing helper methods to perform the type dispatch declaratively, but in these cases it seemed unnatural to do so. The bulk of the remaining casts are related to issues other than event dispatch, for example the lack of parametric polymorphism in Java 1.4.

Table 1 also includes the time to compile each discovery service implementation. We compiled the Java version with Polyglot and the JPred version with our extension to Polyglot, measuring the time to output Java source in each case. The number in each column is the real time averaged over five runs of the compilers on a lightly loaded, modern PC running Linux. Polyglot takes 4.5 seconds to parse and typecheck the Java version and output equivalent code. While the JPred version is slower to compile than the Java version, in absolute terms the JPred version is still quite practical, at 8.6 seconds. Both versions require an additional 1.4 seconds to compile the resulting Java source to bytecode using Sun's `javac` compiler.

The increase in compile time is largely due to time spent in the automatic theorem prover. This case study makes particularly heavy use of predicate dispatch, requiring 452 queries to CVC Lite.

```
protected boolean handle1(Event e) {
  if (state != ACTIVE) {
    ...
    return true;
  }
  if (e instanceof RemoteEvent) ...
}
```

Figure 24: The need for negation predicates in the case study.

```
protected boolean handle1(Event e)
  when state != ACTIVE {
  ...
  return true;
}
protected boolean handle1(Event e)
  when state == ACTIVE {
  return handleActive(e);
}
```

Figure 25: The translation of the code in figure 24 to JPred.

There are several opportunities for reducing the number of queries that we have not yet explored. For example, when computing an overriding relationship between two methods m_1 and m_3, if m_1 is already known to override m_2 and m_2 is already known to override m_3, then we are guaranteed that m_1 overrides m_3 so there is no need to ask the theorem prover. Also, CVC Lite is optimized for handling large formulas with complex boolean structures; other automatic theorem provers, such as Simplify [18], may be better suited for the kinds of small queries that we pose [5]. Finally, despite the increase in compile time, JPred's modular class-by-class compilation style should allow compilation to scale well with program size.

6.3.2 Expressiveness

Figure 21 shows event handlers that employ conjunctions of specializer expressions and equality tests against constants. Many of JPred's other idioms are also utilized in this case study. For example, the portion of figure 22 that is elided by an ellipsis looks as follows:

```
if (eev.x instanceof LeaseDeniedException ||
    eev.x instanceof LeaseRevokedException ||
    eev.x instanceof ResourceRevokedException ||
    eev.x instanceof UnknownResourceException) {
  ...
}
```

JPred's disjunction predicate allows this event dispatch to be declaratively specified.

JPred's negation predicate is also used several times. For example, the event-handling code from `DiscoveryServer` shown in figure 24 naturally requires negation predicates. The handler maintains an integer field that records the handler's "state," one of several constants represented by static, final fields (e.g., `ACTIVE`, `INACTIVE`, `CLOSING`). If the handler is not active, then the action to be performed does not depend on the passed event. Otherwise, event dispatch is performed via a large `if` statement as usual. The JPred version of this code is shown in figure 25. The `handle1` generic

```
...
ServerEntry entry = ...;
if (null != entry) {
  if (ae.capacity >= 0) {
    ...
  } else if (ae.capacity ==
             DiscoveryServer.ANNOUNCE_CLOSING) {
    ...
  } else if (ae.capacity ==
             DiscoveryServer.ANNOUNCE_CLOSED) {
    ...
  }
} else {
  if (ae.capacity >= 0) {
    ...
  } else {
    ...
  }
}
```

Figure 26: Null dispatch and linear arithmetic in the case study.

function dispatches on `state`, invoking `handleActive` to dispatch on the event (via several predicate methods) if the state is active.

Other idioms, including null dispatch, linear arithmetic, and alias dispatch, are also necessary for the case study. An example of the first two idioms occurs in `DiscoveryClient`'s `MainHandler`, in place of the second-to-last ellipsis in figure 20. After declaring and initializing a variable `entry`, both `entry` and the `AnnounceEvent` `ae` are dispatched upon, as shown in figure 26. JPred's predicate language is expressive enough to allow this code to be modularized into its conceptual handlers.

Of the 87 new methods in the JPred implementation of the discovery service, only 30 of them either have no predicate or perform MultiJava-style multimethod dispatch. Therefore, MultiJava is unable to express 57 of the methods, or 66%. Of these 57 methods, 46 of them consist of conjunctions of formal and field dispatches, where each dispatch is either a runtime type test or an equality comparison against a constant. Although MultiJava cannot express these methods directly, it can simulate them by creating appropriate helper methods to perform the dispatch, as shown in figure 23. However, in many cases, MultiJava would require multiple helper generic functions to properly simulate a single JPred method, making MultiJava's solution tedious and unnatural. The final 11 methods, or 13% of the total, rely on predicates that cannot be declaratively expressed in MultiJava, even with unlimited helper methods.

A few expressiveness limitations of JPred arose in the course of the case study. First, there was one case where disjunction would have been used but our rule that conservatively disallows identifier bindings from escaping disjunction was too restrictive. It would be straightforward to extend JPred to resolve this problem. Second, there were two situations where the textual order of `if` cases in the Java implementation was useful. The example in figure 24 illustrates one of these situations; the other is analogous. The handler's state is tested before commencing dispatch on the event. We implemented this ordering constraint by introducing the `handleActive` helper generic function, as shown in figure 25. Unlike the helper method illustrated in figure 23, which is used to work around a syntactic limitation of MultiJava, the `handleActive` helper generic function serves to make explicit a semantic asymmetry in the dispatch logic. It is possible to do away with `handleActive`, dispatching directly on the given event in `handle1` methods, but this requires conjoining the predicate `state == ACTIVE` to the predicate of each such method, which is tedious and harder to understand than the current solution.

```
...
else if (e instanceof ExceptionalEvent) {
  ExceptionalEvent ee = (ExceptionalEvent) e;
  if (ee.x instanceof LeaseRevokedException) {
    return true;
  }
  ... // several more branches of the if statement
  } else if (ee.x instanceof LeaseRevokedException) {
    // This wasn't the bug since it would have complained.
    // FIXME: handle this right
  }
  ...
}
...
return false;
```

Figure 27: An ambiguity found in the discovery service.

Third, there was one situation where it would have been natural to put a method invocation in a predicate, but JPred does not allow this. Finally, there were two cases of event dispatch that did not occur at "top level" in a method. One of these is shown in figure 26. Because there are several statements (represented by the initial ellipsis in the figure) before the dispatch code, a helper generic function `handleAnnounceEvent` was created in the JPred implementation, which is invoked after those statements in order to dispatch on `entry` and `ae`.

6.3.3 Errors Found

During the course of the case study, we found ambiguity errors, nonexhaustive errors, and cast failures. We discuss each in turn.

One redundancy was found during the case study. An ambiguity error was signaled by the JPred typechecker when the `handle1` method of `DiscoveryClient`'s `InputHandler` was rewritten to use JPred. This method is the largest of the 20 methods we translated, at 181 LOC. The relevant code snippet is shown in figure 27. Not only are there two handlers for the case when the event is an `ExceptionalEvent` whose `x` field is a `LeaseRevokedException`, but the handlers have different behavior. The first handler returns true, while the second one falls through and eventually returns false. The method is so complex that even this simplest kind of ambiguity was not caught by the original implementers. The comment in the second handler is in the original code and suggests an unsuccessful attempt to find a related error, most likely without realizing that the second handler is redundant and can therefore never be executed. The author of this code unfortunately does not recall the circumstances.

Three potential nonexhaustive errors were found during the case study. These correspond to situations where a target is assumed to have one of a finite number of runtime types or values, and no default case is provided to handle situations when this assumption is false. One example is shown in figure 26, which was described earlier. The code does not handle the case when `entry` is nonnull and `ae.capacity` is a negative number other than the two constants that are explicitly tested. This nonexhaustive error was found automatically by JPred when typechecking the `handleAnnounceEvent` generic function. Of course, the original programmer could have known about the potential error and simply decided that the missing scenarios were impossible. However, ignoring these scenarios makes the code brittle in the face of changes to the surround-

```
...
else if (e instanceof LeaseEvent) {
  LeaseEvent le = (LeaseEvent) e;
  switch(le.type) {
    ...
    case LeaseEvent.CANCELED:
      ...
      LocalClosure lc = (LocalClosure) le.closure;
      ...
  }
}
```

Figure 28: An unprotected cast found in the discovery service.

ing system; such changes can occur frequently and dynamically in *one.world* applications. JPred forces the programmer to explicitly address all possible scenarios.

In some sense, all of the original handlers in the Java implementation are already guaranteed to be exhaustively implemented, since the Java typechecker ensures that `handle1` returns a boolean value on all feasible paths. However, the Java style of event handling encourages shortcuts that make this checking insufficient. In our example, all the handlers corresponding to the code in figure 26 share a single `return true;` statement. Therefore, the missing case also returns true, even though it should in fact return false to indicate that the event could not be handled. In JPred, each handler is naturally defined in its own method that must explicitly return true or false, so there is no incentive to take such a shortcut.

Two of the three nonexhaustive errors were found automatically by the JPred typechecker when typechecking a helper method. One of these helper methods, `handleAnnounceEvent`, was introduced by us during the case study, as described above. The other helper method existed in the original code. The third nonexhaustive error was detected when we realized that the natural JPred implementation of one event handler returned false in "error cases," whereas the original code returned true. Even in cases where the original handler correctly returned false to handle erroneous situations, the JPred version makes it much easier to understand exactly what conditions correspond to an erroneous situation.

Lastly, several unprotected casts were discovered during the case study. Typically unprotected casts arose when the programmer assumed a correlation between the properties of two different targets. An example from `DiscoveryClient` is shown in figure 28. It is assumed that when the `LeaseEvent`'s `type` is `LeaseEvent.CANCELED`, the event's `closure` is a `LocalClosure`. If that is not the case, a dynamic cast failure will result. In the case study, we removed 11 unprotected casts from the discovery service. JPred's style encourages such dispatch assumptions to be documented in a method's header and makes it natural to do so.

6.3.4 Handler Reuse

In addition to making *one.world* services like the discovery service more understandable and reliable, JPred opens up new possibilities for handler reuse. An example appears in the implementation of `AbstractHandler`. As mentioned earlier, `AbstractHandler`'s `handle` method invokes the abstract method `handle1`, which all subclasses must implement. This structure is used precisely because the Java style of event handling makes handler inheritance awkward. The `handle1` helper method allows subclasses to "inherit" from `AbstractHandler` the functionality for handling erroneous scenarios, as shown in figure 29.

The original developers could have done away with `handle1`,

```
public void handle(Event e) {
  if (handle1(e)) {
    return;
  }
  if (e instanceof ExceptionalEvent) {
    ...
  } else {
    ... // handle unexpected events
  }
}
```

Figure 29: `AbstractHandler`**'s** `handle` **method.**

```
public void handle(Event@ExceptionalEvent e) {
  ...
}
public void handle(Event e) {
  ... // handle unexpected events
}
```

Figure 30: `AbstractHandler`**'s** `handle` **method in JPred.**

removed the first three lines from the `handle` method in figure 29, and required subclasses to simply override `handle`. However, in Java this design would require each subclass to use `super` to explicitly invoke the superclass handler whenever none of the subclass's handlers is applicable, which is tedious and could easily be accidentally omitted. Even with the `handle1` helper generic function, subclasses must still explicitly return false to invoke the inherited error handler, but the Java typechecker helps to ensure that this is not forgotten.

In contrast, JPred can support the desired handler inheritance in the natural way, as shown in figure 30. The `handle1` generic function is no longer necessary. `AbstractHandler` provides two handlers, one for `ExceptionalEvent`s and another for unexpected events. These handlers are implicitly inherited by subclasses, who can add new `handle` methods for particular scenarios of interest. The inherited handlers are automatically dispatched to whenever no method in a subclass is applicable; there is no need for the subclass to explicitly invoke inherited handlers by either invoking `super` or returning false. This design in JPred easily generalizes to support deep hierarchies of handlers with fine-grained code reuse, an idiom which is too unwieldy to consider in Java.

7. RELATED WORK

There have been several previous languages containing a form of predicate dispatch. The original work by Ernst *et al.* [20] was discussed in the introduction and throughout the paper. They implemented predicate dispatch in an interpreter for Dubious [39], a simple core language for formal study of multimethod-based languages. That implementation did not include their static type system.

The predicate language of Ernst *et al.* is more general than ours, including arbitrary boolean expressions from the underlying host language. They also support *predicate abstractions*, which are predicate expressions that are given a name and then referred to in method predicates by name. However, their algorithms for reasoning about predicates only precisely handle propositional logic and specializer expressions, treating all other kinds of predicates as black boxes that are related only by AST equivalence. This sub-

stantially limits the ways in which their predicate language can be used. For example, two methods with predicates $x == 3$ and $x == 4$ would be considered ambiguous. In contrast, JPred's use of off-the-shelf decision procedures supports precise reasoning over JPred's predicate language. Finally, as mentioned in section 3, the static type system described by Ernst *et al.* is global while ours retains Java's modular typechecking strategy.

Ucko [54] describes an extension of the Common Lisp Object System (CLOS) [51, 24] to support predicate dispatch. Similar to the work of Ernst *et al.*, arbitrary Lisp expressions are allowed as predicates. Again a special-purpose algorithm is used for checking validity of predicates. The algorithm is not described in detail, but it appears to only precisely handle propositional logic, specializer expressions, and equality against constants. Static typechecking is not supported. Ucko applies predicate dispatch to enhance the extensibility of an existing computer algebra system written in CLOS. He shows how predicate dispatch is used in the enhanced system to implement symbolic integration and another mathematical function.

Fred [47] is a language that unifies predicate dispatch with features of aspect-oriented programming (AOP) [33]. Like predicate dispatch, methods have predicates associated with them, and logical implication determines method overriding. Like AOP, there is a notion of an "around" method, which is a special method that is always considered to override non-around methods, thereby supporting the addition of new crosscutting code. The language is implemented as a library extension to the MzScheme [42] implementation of Scheme [1]. Similar to the two languages described above, a special-purpose validity checking algorithm is used, which handles propositional logic, specializer expressions, and a limited form of equality. There is no static type system. Instead, the language reports method lookup errors dynamically.

Chambers and Chen [14] describe an algorithm to construct efficient dispatch functions for predicate dispatch. The algorithm computes a directed acyclic graph (DAG) called a *lookup DAG*, which determines the order in which targets are tested. Each node of the lookup DAG is in turn implemented by a decision tree, which determines the order of tests to be performed on a given target (e.g., test that a target is an instance of C_1 before testing that it's an instance of C_2). The authors show performance improvements of up to 30% on a collection of large Cecil programs. Our compilation strategy likely generates less efficient code than the algorithm of Chambers and Chen. At the same time, our strategy interacts well with Java's modular compilation strategy, and it is essentially what a Java programmer would write by hand. Although Chambers and Chen describe their algorithm as a global one, it could probably be adapted to replace our per-class dispatch methods.

Predicate classes [12] are a precursor to predicate dispatch that allow an instance of a class C to be considered to be an instance of some subclass D whenever a given predicate is true. Methods may dispatch on D, and this has the effect of dispatching on whether or not a C instance satisfies D's predicate. Predicate dispatch is more general, allowing predicates that relate multiple arguments to a method. Logical implication is used to determine the subclass relation among predicate classes, analogous with the use of logical implication to determine method overriding in predicate dispatch. However, predicate classes require the implication relationships among predicates to be explicitly declared by the programmer. Similarly, the programmer must declare information about other relationships among predicates, such as disjointness, for use in static typechecking. Classifiers [31] provide similar capabilities to predicate classes, but they use the textual order of methods to define the overriding relation, as in functional languages.

Objective Caml (OCaml) [49, 45] and Scala [50] both extend ordinary ML-style pattern matching with the ability to test an arbitrary boolean expression guard. Unlike JPred, which unifies predicate dispatch with Java's existing dynamic dispatch, pattern matching in OCaml and Scala is independent of the OO dispatch mechanism provided by each of the languages. In both languages, the set of cases in a pattern-matching expression is not extensible, and the textual order of cases defines the overriding relation. Scala does not appear to statically check for nonexhaustive errors or redundancies. OCaml's typechecker does check for nonexhaustive errors, but it does so simply by assuming that every predicate guard could evaluate to false simultaneously. This has the effect of statically signaling a warning whenever there does not exist a default case (i.e., a case without a predicate). In contrast, JPred sometimes does not require default methods. Unlike in JPred, pattern matching in OCaml and Scala is a first-class expression and need not appear at the top level of a function's implementation.

JMatch [36] extends Java with a sophisticated form of pattern matching that includes many of the idioms supported by JPred, including type dispatch, dispatch on fields, and identifier binding. JMatch's patterns additionally provide support for expressing iteration, for defining abstract patterns that hide a value's underlying representation, and for bidirectional computation as provided by logic programming languages. As with OCaml and Scala, pattern matching in JMatch has a functional style: pattern matching is separate from Java's OO dispatch mechanism, pattern matching is a first-class expression, the cases in a pattern-match expression are not extensible, and the textual order of cases defines the overriding relation. JMatch does not statically check for nonexhaustive errors or redundancies.

Several recent languages, including XStatic [26], CDuce [6], and HydroJ [35], support pattern matching for XML-like [9] data. The patterns in these languages overlap with JPred's predicate language; for example, JPred's disjunction predicate corresponds to union patterns for XML data. However, each can express things that the other cannot. The XML languages lack support for relational and arithmetic predicates as well as predicates that relate multiple arguments to a function. JPred lacks support for arbitrary regular expressions. Most of the languages proposed for manipulating XML data are based on functional languages, and their pattern matching constructs therefore have the same style as pattern matching in OCaml, Scala, and JMatch. An exception is HydroJ, an extension of Java with support for XML data. HydroJ unifies XML pattern matching with Java's OO dispatch and uses a form of predicate implication as the overriding relation. Like JPred, HydroJ adapts our prior work on modular typechecking of multimethods [38] to support modular typechecking of patterns.

Our previous languages MultiJava [16] and Extensible ML (EML) [38] support modular typechecking in the presence of multimethods. MultiJava's multimethod dispatch can be viewed as the subset of JPred supporting only conjunctions of specializer expressions and equality tests against constants, and only for formal parameters. MultiJava also supports *open classes*, the ability to add new methods to existing classes noninvasively. JPred does not support open classes, but it would be straightforward to add them. EML subsumes MultiJava's predicate language, additionally containing the ML-style pattern matching idioms of identifier binding and dispatch on substructure. JPred extends EML's pattern language to include disjunction and negation, arbitrary equality and other relational predicates, linear arithmetic, and predicates that relate multiple arguments, while retaining modular typechecking and compilation. JPred safely relaxes the modularity requirements of MultiJava and EML, for example not always requiring a default method. This

relaxation allows new programming idioms to be expressed, including partially abstract methods. JPred's use of off-the-shelf decision procedures is also novel.

8. CONCLUSIONS AND FUTURE WORK

We have described JPred, a practical design and implementation of predicate dispatch for Java. JPred naturally augments Java while retaining its modular typechecking and compilation strategies. This contrasts with the global typechecking and compilation algorithms of prior languages containing predicate dispatch. JPred uses off-the-shelf decision procedures to reason about predicates, both for determining the method overriding relation and for static exhaustiveness and ambiguity checking. This contrasts with the special-purpose and overly-conservative algorithms for reasoning about predicates that are used by prior languages with predicate dispatch. We presented a case study illustrating the utility of JPred on an existing Java application, including its use in the detection of several errors.

JPred could be extended in several ways. The predicate language currently only supports literals of integer and boolean type. It would be straightforward to support the other Java literals as well as arrays, along with many of their associated primitive operations. Named predicate abstractions could be convenient and would not cause any technical problems. The case study identified a few ways to extend our predicate language, as described in section 6.3.2. These include relaxing the rules for when identifiers can escape from disjunction and supporting method calls in predicates. To allow method calls in predicates, it may be necessary to introduce a `pure` modifier, to declare a method to be side-effect-free. Similar to method calls, it may be useful to incorporate a notion of *views* [55] in JPred. This would allow a class to export a virtual representation to be dispatched upon by clients, as an alternative to allowing clients to dispatch directly on fields. Finally, a notion of `resend` [13, 37], which generalizes Java's `super` to walk up JPred's method-overriding partial order, could be useful to allow predicate methods within a class to easily share code.

9. ACKNOWLEDGMENTS

Thanks to Sergey Berezin and Stephen Chong for their incredibly responsive support regarding CVC Lite and Polyglot, respectively. Thanks to Eric Lemar for answering questions about the *one.world* code used in the case study. Thanks to Dan Grossman for pointers and discussion about related work. Thanks to Craig Chambers, Mike Ernst, Robert Grimm, Keunwoo Lee, and Jens Palsberg for helpful comments on the paper. Finally, thanks to the anonymous reviewers for their useful and detailed feedback, including the idea of allowing a set of resolving methods instead of a single one.

10. REFERENCES

[1] H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. A. IV, D. P. Friedman, E. Kohlbecker, G. L. Steele Jr., D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, and M. Wand. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, Aug. 1998.

[2] D. Ancona, G. Lagorio, and E. Zucca. A formal framework for Java separate compilation. In *Proceedings of the 2002 European Conference on Object-Oriented Programming*, LNCS 2374, Malaga, Spain, June 2002. Springer-Verlag.

[3] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language Third Edition*. Addison-Wesley, Reading, MA, third edition, 2000.

[4] M. G. Assaad and G. T. Leavens. Alias-free parameters in C for better reasoning and optimization. Technical Report 01-11, Department of Computer Science, Iowa State University, Ames, Iowa, Nov. 2001.

[5] C. Barrett. Personal communication, Mar. 2004.

[6] V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-centric general-purpose language. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 51–63. ACM Press, 2003.

[7] D. Blei, C. Harrelson, R. Jhala, R. Majumdar, G. C. Necula, S. P. Rahul, W. Weimer, and D. Weitz. Vampyre: A Proof Generating Theorem Prover. `http://www.cs.ucla.edu/~rupak/Vampyre`.

[8] D. G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. CommonLoops: Merging Lisp and object-oriented programming. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 17–29. ACM Press, 1986.

[9] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. eXtensible Markup Language (XML) 1.0 (Third Edition). World Wide Web Consortium recommendation, `http://www.w3.org/TR/REC-xml`, 2004.

[10] G. Castagna. Covariance and contravariance: Conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, Mar. 1995.

[11] C. Chambers. Object-oriented multi-methods in Cecil. In O. L. Madsen, editor, *Proceedings ECOOP '92*, LNCS 615, pages 33–56. Springer-Verlag, June 1992.

[12] C. Chambers. Predicate classes. In *Proceedings of the 7th European Conference on Object-Oriented Programming*, pages 268–296, July 1993.

[13] C. Chambers. The Cecil language specification and rationale: Version 2.1. `www.cs.washington.edu/research/projects/cecil/pubs/cecil-spec.html`, Mar. 1997.

[14] C. Chambers and W. Chen. Efficient multiple and predicate dispatching. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 238–255, Denver, Colorado, 1999. ACM Press.

[15] C. Clifton. MultiJava: Design, implementation, and evaluation of a Java-compatible language supporting modular open classes and symmetric multiple dispatch. Technical Report 01-10, Department of Computer Science, Iowa State University, Ames, Iowa, Nov. 2001.

[16] 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*, pages 130–145, Minneapolis, Minnesota, Oct. 2000.

[17] CVC Lite home page. `http://verify.stanford.edu/CVCL`.

[18] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, 2003.

[19] S. Drossopoulou, S. Eisenbach, and D. Wragg. A fragment calculus — towards a model of separate compilation, linking and binary compatibility. In *Logic in Computer Science*,

pages 147–156, 1999.

[20] M. Ernst, C. Kaplan, and C. Chambers. Predicate dispatching: A unified theory of dispatch. In E. Jul, editor, *ECOOP '98–Object-Oriented Programming*, LNCS 1445, pages 186–211. Springer, 1998.

[21] M. Fahndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 13–24. ACM Press, 2002.

[22] C. Flanagan, R. Joshi, X. Ou, and J. B. Saxe. Theorem proving using lazy proof explication. In *CAV: International Conference on Computer Aided Verification*, Boulder, Colorado, USA, July 2003.

[23] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 1–12. ACM Press, 2002.

[24] R. P. Gabriel, J. L. White, and D. G. Bobrow. CLOS: Integrating Object-Oriented and Functional Programming. *Commun. ACM*, 34(9):28–38, Sept. 1991.

[25] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts, 1995.

[26] V. Gapeyev and B. C. Pierce. Regular object types. In *Proceedings of the 2003 European Conference on Object-Oriented Programming*, LNCS 2743, Darmstadt, Germany, July 2003. Springer-Verlag.

[27] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11. ACM Press, 2003.

[28] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, Boston, Mass., 2000.

[29] R. Grimm. *System Support for Pervasive Applications*. Ph.D. dissertation, Department of Computer Science & Engineering, University of Washington, 2002.

[30] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, pages 93–104, 2000.

[31] J. G. Hosking, J. Hamer, and W. Mugridge. Integrating functional and object-oriented programming. In *Proceedings of the 1990 TOOLS Pacific*, pages 345–355, 1990.

[32] D. H. H. Ingalls. A simple technique for handling multiple polymorphism. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 347–349. ACM Press, 1986.

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

[34] G. T. Leavens and O. Antropova. ACL — Eliminating parameter aliasing with dynamic dispatch. Technical Report 98-08a, Department of Computer Science, Iowa State University, Ames, Iowa, Feb. 1999.

[35] K. Lee, A. LaMarca, and C. Chambers. HydroJ: Object-oriented pattern matching for evolvable distributed systems. In *Proceedings of the 2003 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Anaheim, CA, Oct. 2003.

[36] J. Liu and A. C. Myers. JMatch: Iterable abstract pattern matching for Java. In V. Dahl and P. Wadler, editors, *Practical Aspects of Declarative Languages, 5th International Symposium*, volume 2562 of *Lecture Notes in Computer Science*, pages 110–127. Springer, 2003.

[37] T. Millstein. *Reconciling Software Extensibility with Modular Program Reasoning*. Ph.D. dissertation, Department of Computer Science & Engineering, University of Washington, 2003.

[38] T. Millstein, C. Bleckner, and C. Chambers. Modular typechecking for hierarchically extensible datatypes and functions. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '02)*, volume 37(9) of *ACM SIGPLAN Notices*, pages 110–122, New York, NY, Sept. 2002. ACM.

[39] T. Millstein and C. Chambers. Modular statically typed multimethods. *Information and Computation*, 175(1):76–118, May 2002.

[40] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[41] D. A. Moon. Object-oriented programming with Flavors. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 1–8. ACM Press, 1986.

[42] MzScheme home page. `http://www.plt-scheme.org/software/mzscheme`.

[43] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.

[44] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *Proceedings of CC 2003: 12'th International Conference on Compiler Construction*. Springer-Verlag, Apr. 2003.

[45] OCaml home page. `http://www.ocaml.org`.

[46] *one.world* home page. `http://cs.nyu.edu/rgrimm/one.world`.

[47] D. Orleans. Incremental programming with extensible decisions. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 56–64. ACM Press, 2002.

[48] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *Proceedings of the 1999 USENIX Annual Technical Conference (USENIX-99)*, pages 199–212, Berkeley, CA, June 6–11 1999. USENIX Association.

[49] D. Rémy and J. Vouillon. Objective ML: An effective object-oriented extension of ML. *Theory and Practice of Object Systems*, 4(1):27–52, 1998.

[50] The Scala language home page. `http://lamp.epfl.ch/~odersky/scala`.

[51] G. L. Steele Jr. *Common Lisp: The Language, Second Edition*. Digital Press, Bedford (MA), USA, 1990.

[52] B. Stroustrup. *The C++ Programming Language: Third Edition*. Addison-Wesley, Reading, Mass., 1997.

[53] A. Stump, C. W. Barrett, and D. L. Dill. CVC: A Cooperating Validity Checker. In E. Brinksma and K. G.

Larsen, editors, *14th International Conference on Computer Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 500–504. Springer-Verlag, 2002. Copenhagen, Denmark.

[54] A. M. Ucko. Predicate Dispatching in the Common Lisp Object System. Technical Report 2001-006, MIT Artificial Intelligence Laboratory, June 2001.

[55] P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 307–313. ACM Press, 1987.

[56] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In G. Ganger, editor, *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 230–243, 2001.