Expressive and Modular Predicate Dispatch for Java

TODD MILLSTEIN, CHRISTOPHER FROST, JASON RYDER
and ALESSANDRO WARTH
University of California, Los Angeles

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 the practical utility of this language feature.

We describe JPred, a backward-compatible extension to Java supporting predicate dispatch. While prior languages with predicate dispatch have been extensions to toy or nonmainstream 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 informally, we present an extension to Featherweight Java that formalizes the language and its modular type system, which we have proven sound. Finally, we discuss two case studies that illustrate the practical utility of JPred, 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; patterns*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Syntax, semantics*

General Terms: Design, Languages

This article incorporates results first presented at OOPSLA 2004 [Millstein 2004] and FOOL/WOOD 2006 [Frost and Millstein 2006].

This material is based upon work supported by the National Science Foundation under Grant Nos. CCF-0427202 and CCF-0545850, as well as by a generous gift from Microsoft Research.

Authors' address: T. Millstein (corresponding author), C. Frost, J. Ryder, and A. Warth, UCLA Computer Science Department, University of California, Los Angeles, CA 90095-1596; email: todd@cs.ucla.edu.

Additional Key Words and Phrases: Predicate dispatch, dynamic dispatch, modular typechecking

ACM Reference Format:

Millstein, T., Frost, C., Ryder, J., and Warth, A. 2009. Expressive and modular predicate dispatch for Java. ACM Trans. Program. Lang. Syst. 31, 2, Article 7 (February 2009), 54 pages. DOI = 10.1145/1462166.1462168 http://doi.acm.org/10.1145/1462166.1462168

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 [Arnold et al. 2005; Gosling et al. 2005] 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 [Chambers 1992, 1997] and MultiJava [Clifton et al. 2000, 2006], 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 [Milner et al. 1997] 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. It does so by alleviating 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 single applicable method overrides all others).

In 1998, Ernst et al. introduced the concept of *predicate dispatch* [Ernst et al. 1998]. 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 [Chambers and Chen 1999; Ucko 2001; Orleans 2002], 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 nonmainstream 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 article 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 [Millstein 2003]. This generalization also allows for modular typechecking in the presence of multiple inheritance (e.g., as supported by Java interfaces), a long-standing problem for multimethod-based languages.

—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 CVC3 [2009], which contains decision procedures for several decidable theories, including propositional logic, rational linear arithmetic, and the theory of equality. CVC3 is sound and complete for validity queries over JPred's predicate language, so our language and type system remain well defined and predictable.

—We formalize JPred's form of predicate dispatch in an extension to Featherweight Java [Igarashi et al. 2001]. This formalism models JPred's modular type system and its usage of decision procedures, and a type soundness theorem proves their sufficiency. To our knowledge, ours is the first provably sound formalization of predicate dispatch.

—We have implemented JPred as an extension in the Polyglot extensible compiler framework for Java [Nystrom et al. 2003]. 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 several case studies using the language, two of which are described here. First, we have rewritten a Java implementation of a discovery service that is part of the one.world platform for pervasive computing [Grimm et al. 2004] to use JPred. Second, we have employed JPred's support for modularly typesafe dispatch on interfaces in the implementation of the JPred compiler. We illustrate and quantify the advantages that JPred provides in both case studies, including its use in the detection of several errors.

The rest of the article is structured as follows. 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 presents Featherweight JPred, which formalizes a core subset of JPred. Section 7 describes the case study illustrating JPred's effectiveness. Section 8 discusses related work, and Section 9 concludes.

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 detailed description of the semantics of method invocation in JPred.

$$
\begin{array}{rcl}
\mathit{pred} & ::= & \mathit{lit} \;\Big|\; \mathit{tgt} \;\Big|\; \mathit{tgt}\texttt{@}\mathit{Type} \\
 & \Big| & \mathit{Identifier}\ \texttt{as}\ \mathit{tgt} \;\Big|\; \mathit{Identifier}\ \texttt{as}\ \mathit{tgt}\texttt{@}\mathit{Type} \\
 & \Big| & \mathit{uop}\ \mathit{pred} \;\Big|\; \mathit{pred}\ \mathit{bop}\ \mathit{pred} \\
\mathit{lit} & ::= & \texttt{null} \;\Big|\; \mathit{IntegerLiteral} \;\Big|\; \mathit{BooleanLiteral} \\
\mathit{tgt} & ::= & \texttt{this} \;\Big|\; \mathit{Identifier} \;\Big|\; \mathit{tgt}\texttt{.}\mathit{Identifier} \;\Big|\; \mathit{tgt}\texttt{[}\mathit{pred}\texttt{]} \\
 & \Big| & \mathit{tgt}\texttt{.}\mathit{Identifier}\texttt{(}\mathit{pred}^{*}\texttt{)} \;\Big|\; \mathit{Identifier}\texttt{(}\mathit{pred}^{*}\texttt{)} \\
\mathit{uop} & ::= & \texttt{!} \;\Big|\; \texttt{-} \\
\mathit{bop} & ::= & \texttt{\&\&} \;\Big|\; \texttt{||} \;\Big|\; \texttt{==} \;\Big|\; \texttt{!=} \;\Big|\; \texttt{<} \;\Big|\; \texttt{<=} \;\Big|\; \texttt{>} \;\Big|\; \texttt{>=} \;\Big|\; \texttt{+} \;\Big|\; \texttt{-} \;\Big|\; \texttt{*}
\end{array}
$$

Fig. 1. The abstract syntax of predicate expressions in JPred. The notation *pred** denotes zero or more comma-separated predicate expressions. Nonterminals *Identifier*, *Type*, *IntegerLiteral*, and *BooleanLiteral* are defined as in the Java Language Specification.

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 comprise a side-effect-free subset of Java expressions of type `boolean` and can consist of literals, references to formals and fields in scope, array accesses, invocations of *pure* methods, dynamic dispatch on types, identifier binding, and a host of Boolean, relational, and arithmetic operations. We call a method containing a `when` clause a *predicate method*.

2.1 Multimethod Dispatch on Classes

2.1.1 *Event-Based Systems.* Figure 2 illustrates an event-based implementation of a file editor in Java. The base class `Event` has a number of subclasses, each representing a different possible action desired by the user. The `handle` operation is invoked when an event is triggered, and 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.

```
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 */ }
  }
}
```

Fig. 2. A file editor implemented in Java.

```
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 */ }
}
```

Fig. 3. The file editor implemented in JPred.

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 this 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* [Bobrow et al. 1986], using JPred's *specializer expression* to dynamically dispatch on their arguments in addition to the receiver. Similar to multimethod notation in Cecil [Chambers 1992, 1997], 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) { ... }
```

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 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 */ }
}
```

Fig. 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.

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 automatically invokes the appropriate handler for each scenario.

Further, the code is now statically type safe. There is no potential for dynamic cast failures. In addition, the JPred typechecker checks that the handlers cover all possible scenarios and are not ambiguous with one another. This check ensures that each dynamic invocation of `handle` will find a single *most-specific* method (according to the method overriding relation) to invoke for the passed event. As described in Section 3, this check is safely performed *modularly*, one compilation unit at a time, along with the usual Java typechecks. The check catches common errors that go undetected in the Java version in Figure 2. For example, if the last `handle` method in `FileEditor` of Figure 3 is omitted, the JPred typechecker signals the following error at compile time, because of the potential for invocations of `handle` that have no applicable method to invoke.

```
FileEditor.pj:2: This method's associated operation is not fully
    implemented.
     One way to resolve the problem is to add a default method, which is a
     method that has no 'when' clause.
     void handle(Event e) when e@Open {}
          ^
```

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 hence they are naturally inherited by subclasses. For example, an extended version of the editor is shown in Figure 4. This editor

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

Fig. 5. Noninvasive visitors in JPred.

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 7 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 [Clifton et al. 2000], so the first `handle` method in Figure 3 can be equivalently written as follows.

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

MultiJava-style specializers are 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 [Millstein 2003]. 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 [Gamma et al. 1995], 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* [Ingalls 1986] 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

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

Fig. 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());
  }
}
```

Fig. 7. Disjunction in JPred predicate expressions.

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.

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 means that specialized types from a method's predicate cannot always 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 [Ernst et al. 1998]. However, the current rule handles the common case and is simple to understand.

2.2 Interface Dispatch and Ordered Dispatch

Java implicitly only supports dynamic dispatch on class types because only classes can contain method implementations. However, the generalization to multiple or predicate dispatch makes it natural to consider dynamic dispatch on all types. JPred's @ predicate supports dispatch on class types, interface types, and array types. The ability to dispatch on types other than classes is often useful in practice. For example, it is common for a framework to expose only interfaces to clients, keeping the underlying implementation classes hidden (e.g., Polyglot [Nystrom et al. 2003] and Eclipse [Eclipse 2007]). Without interface dispatch, clients of these frameworks cannot enjoy the benefits of multiple dispatch.

Interface dispatch poses a special challenge for modular typechecking of multiple dispatch, because interfaces support multiple inheritance. This subsection describes the problem and JPred's solution.

2.2.1 *Interface Dispatch and Modular Typechecking.* Consider again the file editor in Figure 3. The fact that handle is modularly typesafe relies critically on the fact that classes in Java support only single inheritance. It is the lack of multiple inheritance that ensures, without knowledge of all classes in the program, that the first and second handle methods are unambiguous. This is because the language prevents the existence of a class that subclasses both Open and SaveAs (assuming neither is a subclass of the other). Consider instead a variant of our event hierarchy in which Java interfaces are used rather than classes. With this new hierarchy, there could exist a class that implements both the Open and SaveAs interfaces. If an instance of such a class were ever passed to handle, there would be no single most-specific method implementation to invoke. Therefore, the JPred typechecker signals an ambiguity between the first two handle methods at compile time.

```
FileEditor.pj:2: This method is ambiguous with the one at FileEditor.pj:3.
    void handle(Event e) when e@Open {}
         ^
```

Because of this potential for ambiguities, prior languages supporting multiple dispatch and modular typechecking have either disallowed interface dispatch entirely [Clifton et al. 2006], severely restricted the usage of multiple inheritance or multiple dispatch [Millstein and Chambers 2002; Baumgartner et al. 2002], or linearized the semantics of multiple inheritance or multiple dispatch [Agrawal et al. 1991; Boyland and Castagna 1997].

2.2.2 *Modularly Typesafe Interface Dispatch.* Our solution to this problem is based on a simple yet powerful observation: While it is impossible to modularly know all the classes (if any) that cause a pair of methods to be ambiguous due to multiple inheritance, the expressiveness of predicate dispatch nonetheless allows programmers to modularly resolve all potential ambiguities. Therefore, we need not impose any restrictions or modifications to either multiple inheritance or predicate dispatch. Instead, we simply leave it to the programmer to resolve any signaled ambiguities in the manner deemed most appropriate.

For example, one way to resolve the ambiguity between the first two methods in Figure 3 is to add the following method.

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

Because this new method overrides both methods and is applicable whenever both of them are applicable, the ambiguity between those two methods can never manifest itself as a runtime ambiguity.

As an alternative to adding a dedicated method to handle the ambiguity, predicate dispatch also allows a programmer to specify one of the original ambiguous methods to be favored in the event of an ambiguity. For example, the programmer could revise the predicate on the second `handle` method to be `e@SaveAs && !e@Open`, thereby explicitly indicating that events implementing both `Open` and `SaveAs` should be dispatched to the first `handle` method; the second method is no longer applicable.

These two approaches to resolving ambiguities naturally generalize beyond the case when exactly one argument (in addition to the receiver) is dispatched upon. In general, given two ambiguous JPred methods with predicates P_1 and P_2, their ambiguity can be resolved by adding a third method whose predicate is P_1 `&&` P_2. Alternatively, the ambiguity can be resolved by modifying the predicate on the second method to instead be P_2 `&& !`P_1 (or by modifying the predicate on the first method symmetrically). We have shown two common ways for programmers to resolve ambiguities, but variations on these approaches are possible.

2.2.3 *Ordered Dispatch.* To mitigate the burden of resolving ambiguities on programmers, we have introduced a natural syntactic sugar for predicate methods. This sugar is inspired by pattern matching in functional languages like ML [Milner et al. 1997], which uses a “first-match” semantics in contrast to the “best-match” semantics typical of OO languages. We observe that the best-match semantics of predicate dispatch, based on predicate implication, is expressive enough to encode the first-match semantics.

Figure 8 illustrates our syntactic sugar, which we refer to as *ordered dispatch*, using a revised version of the `FileEditor`. Ordered dispatch consists of a single method declaration with several associated *cases*. Conceptually, ordered dispatch uses a first-match semantics: Upon a message send, each case’s predicate is tested one-by-one in textual order, and the first case whose predicate is satisfied is invoked. Given this semantics, the code in Figure 8 is modularly typesafe even if the events are interfaces. For example, if the passed event implements both `Open` and `SaveAs`, the first case in the figure will be invoked.

```
class FileEditor {
  void handle(Event e)
    when e@Open { ... }
  | when e@SaveAs { ... }
  | when e@Save { ... }
  | { ... }
}
```

Fig. 8. Ordered dispatch in JPred.

Unlike prior approaches that resolve ambiguities using textual order [Boyland and Castagna 1997], the introduction of ordered dispatch does not entail any modifications to JPred's method-lookup semantics, since ordered dispatch is purely syntactic sugar. An ordered dispatch of the form

$\texttt{T m(}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{) when P}_1\ \texttt{\{ ... \} |}\ \cdots\ \texttt{| when P}_n\ \texttt{\{ ... \}}$

is desugared by the JPred compiler to the following collection of regular JPred methods, whose textual order is irrelevant.

$\texttt{T m(}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{) when P}_1\ \texttt{\{ ... \}}$
$\texttt{T m(}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{) when P}_2\ \texttt{\&\& !P}_1\ \texttt{\{ ... \}}$
...
$\texttt{T m(}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{) when P}_n\ \texttt{\&\& !P}_1\ \texttt{\&\&}\ \cdots\ \texttt{\&\& !P}_{n-1}\ \texttt{\{ ... \}}$

Therefore, programmers can also easily mix ordered dispatch with regular JPred methods. As a simple example, in Figure 8 the programmer could choose to make the last case in the ordered dispatch declaration a separate JPred method.

2.3 Field Dispatch, Array Access, and Identifier Binding

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 9. The example also illustrates another use of disjunction in predicates.

As another example, Figure 10 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.

```
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 */ }
}
```

Fig. 9. Field dispatch in JPred.

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

Fig. 10. Another example of field dispatch.

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

Fig. 11. Narrowing the static type of a field is unsound.

JPred also allows predicates to dispatch on the elements of an array, which is conceptually a form of field dispatch for arrays. As a simple example, suppose our `DataNode` class were revised to represent a node of an *n*-ary tree, replacing the `left` and `right` fields with a `children` field of type `TreeNode[]`. In this case, the predicate on the first `getMin` method in Figure 10 would be changed to the predicate `this.children[0]@EmptyNode` in order to test whether the leftmost child is empty. As in Java, JPred performs no static array-bounds checking, instead deferring such checking to runtime.

Unlike specialized types for formals, specialized types for fields and array elements 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 10, even though the method can only be invoked when `this.left` is an instance of `EmptyNode`. The `unsound` method in Figure 11 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 previous example, an invocation `dn.unsound(dn)` causes `this.left` and `n.left` to have the same *l-value*, so assigning to one of them implicitly causes the value of the other to change. By forcing a field expression to retain its original static type, JPred ensures type soundness, regardless of how the expression's value is updated through aliases. In JPred, the code in Figure 11 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 and local variables in Java do not suffer from this aliasing problem: The only way to make a variable x refer to a different object is by assigning to x itself. 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 and array elements 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 have the same semantics as local variables, so 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 11 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 have the same l-value, these expressions do not have the same l-value as `l`. Therefore, assigning to `n.left` does not affect the value of `l`. 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 equivalent to the predicate `true`, since it always succeeds (modulo null dereferences and array-bounds violations, which JPred, like Java, does not statically prevent).

2.4 Equality

Functional languages like ML allow formals and (the analog of) fields to be tested against constants via pattern matching. JPred can express this idiom via equality testing against literals and other compile-time constant expressions. For example, `FileProtocol` in Figure 12 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.

```
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 */ }
}
```

Fig. 12. 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; }
}
```

Fig. 13. Extending a finite-state machine in JPred.

Unlike a corresponding implementation with functional-style pattern matching, the FSM in Figure 12 is easily extensible. For example, Figure 13 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. 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@T` is false for every type `T`.) 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 analog in functional-style pattern matching, since JPred allows targets to be compared against one another. An example is shown in Figure 14, where the `equals` method inherited from `Object` is overridden. The first method's predicate shows that JPred subsumes *alias dispatch* [Leavens and Antropova 1999; Assaad and Leavens 2001], in which method 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

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

Fig. 14. 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); }
}
```

Fig. 15. Linear inequalities in JPred.

library. 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).

2.5 Linear Arithmetic and Partially Abstract Methods

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 [Gosling et al. 2005]. Forcing arithmetic to be linear ensures that testing relationships among predicates, such as logical implication, remains decidable.

Figure 15 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. Further, JPred's static typechecker is able to verify that the three methods are both exhaustive and unambiguous, ensuring a single most-specific method to invoke for every possible type-correct argument value.

As another example, consider an `nth` method on tree nodes, 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 16 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

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

Fig. 16. Partially abstract methods in JPred.

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.

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. Many other operations besides `nth` naturally have error scenarios and hence would also benefit from being partially abstract.

2.6 Pure Methods

It is often useful to invoke methods within a predicate expression. For example, a class's fields are typically kept `private`, so clients of the class must access these fields indirectly through getter methods. Therefore, clients who wish to dispatch on the values of fields must be able to invoke these getter methods in predicates. As another example, although the == operator can be used to compare objects for pointer equality in a predicate expression, as shown earlier, it is typically desirable to compare objects for logical equality using the `equals` method from `java.lang.Object`.

JPred therefore allows methods to be invoked within predicate expressions. However, JPred requires such methods to be side-effect-free. Side effects in predicates would cause several problems. Because JPred makes no guarantees about the order in which predicates are evaluated, it would be difficult for programmers to understand the impact of any side effects in predicates. Similarly, side effects could cause the evaluation of a predicate expression to affect the values of other predicate expressions (including itself). This could lead to a situation where the method selected by dynamic dispatch is invoked in a context where its associated predicate expression no longer holds.[1]

To enforce the lack of side effects, JPred requires any method invoked within a predicate expression to have been declared `pure`. The static typechecker ensures that each method declared `pure` is side-effect-free. Our current implementation uses a simple set of rules that is easy to understand. First, a `pure` method may not assign to fields or to array elements; assignment to local variables is

[1]Unfortunately, it is possible for such a situation to arise even without side effects in predicates, in the presence of multithreading.

allowed freely. Second, a `pure` method may not invoke non-`pure` methods. Third, a `pure` method may not invoke any constructors. Finally, since it is not possible to statically know which method implementation is invoked upon a message send, any method that overrides a `pure` method must itself be declared `pure`.

Although these rules are fairly restrictive, they support a variety of useful kinds of method calls in predicates, including the two scenarios mentioned before. For example, the implementer of the `BinaryExpr` AST node used in the typechecking visitor of Figure 9 could make the `operator` field `private`, instead providing an appropriate accessor method.

```
public pure Operator getOperator() { return this.operator; }
```

In that case, the typechecking visitor can access the node's operator via the invocation `n.getOperator()`. As another example, the two `equals` methods in Figure 14 could be declared `pure` and thereby employed in predicates. Others have proposed more precise analyses for ensuring purity of Java methods (e.g., Salcianu and Rinard [2005]), which we could adapt if desired.

2.7 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 *method family* [Clifton et al. 2006] (a collection of methods of the same name, number of arguments, and static argument types) will be invoked dynamically.[2]

At runtime, each expression e_i is evaluated to produce a value v_i and the single most-specific applicable method belonging to the statically determined method family 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 single most-specific applicable method if it is the only applicable method that *overrides* all other applicable methods. Finally, one method m_1 overrides another method m_2 if either of the following holds.

—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

[2]A method family has also been known as a *generic function* [Bobrow et al. 1986; Moon 1986], but we avoid that terminology to prevent confusion with Java 1.5's notion of generics.

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

Fig. 17. A problem with the symmetric approach to method lookup.

methods in the figure are applicable, and the second method is the single 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 single 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 [Castagna 1995]. 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 single 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 17. 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 17. 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 a method whose predicate is logically false) is ambiguous with `C`'s `m` method.

3. STATIC TYPECHECKING

This section informally 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. The section ends with a discussion of the interaction between predicate methods and Java's *generics*.

3.1 Typechecking Message Sends

Message sends are typechecked in JPred exactly as in Java; no modifications are required. As mentioned in Section 2.7, Java's static overload resolution is performed to determine which method family 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 method family 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 method family's methods are *exhaustive*, so that message-not-understood errors cannot happen at runtime. Finally, we describe the checks ensuring that a method family'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 new 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 are typechecked exactly as they are in Java. Arithmetic predicate expressions are additionally checked to be linear, as described in Section 2.5.

Specializer expressions of the form *tgt*`@`*Type* are typechecked like Java's `instanceof` expression. Namely, the static type of *tgt* must obey the rules for *casting conversion* to *Type* [Gosling et al. 2005], which ensure that it is possible for the runtime class of *tgt* to be a subtype of *Type*. Specializer expressions of the form *Identifier* `as` *tgt*`@`*Type* are typechecked identically, and additionally the specialized name *Identifier* is given the static type *Type*. 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 16. 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 access modifiers and declared exceptions as all other methods belonging to the same method family 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 12 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.

Finally, the typechecker ensures that a method's associated predicate is *satisfiable*, meaning that there is at least one possible context in which the predicate evaluates to true. A predicate is determined to be unsatisfiable if its negation is logically valid. This check is not required for soundness but it provides useful feedback for programmers. Since a method with an unsatisfiable predicate can never be invoked, the predicate likely is the result of a programmer error.

The check for predicate satisfiability is particularly useful for methods that employ the ordered dispatch syntactic sugar discussed in Section 2.2.3. In this context, the satisfiability check plays exactly the role of the *match redundant* warnings provided by languages like Standard ML [Milner et al. 1997], which warn about unreachable cases of a function. For example, suppose the order of the second and third cases were swapped in Figure 8. Assuming that `SaveAs` is a subtype of `Save`, the `SaveAs` case would now be unreachable. The JPred typechecker correctly signals an error, because the desugared version of the predicate `e@SaveAs` is now `e@SaveAs && !e@Open && !e@Save`, which is unsatisfiable.

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 15 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 16 for all nonnegative integer arguments. Finally, in JPred we must check that *C* fully implements any new method families 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` method family for all possible pairs of argument events.

All of these checks are performed in a uniform way. To check exhaustiveness of a method family from a class *C*, we collect all of the concrete methods of that method family 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 15. 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`. Indeed, these events may not even have been implemented when `FileEditor` is typechecked and compiled.

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 preceding `contains` example. Ernst et al.'s exhaustiveness checking algorithm for predicate dispatch [Ernst et al. 1998] safely requires fewer default methods than JPred, but the algorithm requires whole-program knowledge.

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 single 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++ [Stroustrup 1997], 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 method family. The algorithm for checking each method pair generalizes our earlier algorithm for modular ambiguity checking in Extensible ML (EML) [Millstein et al. 2004] 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 12. 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 new method is ambiguous with the first `check` method in Figure 12: 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 { ... }
```

```
class Generic<T> {
  void m1(List<T> l) { ... }
  void m1(List<T> l) when l@ArrayList<T> { ... }

  void m2(T t) { ... }
  void m2(T t) when t@String { ... }
}
```

Fig. 18. Predicate dispatch and generics.

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 2.7 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 similar to the original ambiguity algorithm for predicate dispatch [Ernst et al. 1998]. However, that algorithm is performed on all pairs of methods belonging to the same method family 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.

3.3 Generics and Predicate Dispatch

Java's generics are naturally handled by the rules just described. The key requirement, mentioned in Section 3.2.1, is that specializer expressions obey Java's rules for casting conversion. In other words, JPred allows specializing an object of type `T1` to the type `T2` if and only if Java allows an object of type `T1` to be cast to `T2`. Figure 18 illustrates some simple examples of predicate methods that typecheck successfully in JPred.

In order to interoperate with legacy code written before the advent of generics, and because generics in Java are implemented via type erasure, casts in Java may be (partially) *unchecked*. An unchecked cast can succeed dynamically even though the given object does not in fact have the cast-to type, thereby potentially causing future runtime type errors. For example, a cast to `ArrayList<String>` dynamically only requires the given list's class to be a subtype of `ArrayList`, regardless of the type of its elements. Later code may then erroneously assume that the list only contains strings. The Java typechecker signals a static warning whenever a program contains an unchecked cast.

As a result, dynamic dispatch in JPred may also be unchecked. For example, suppose we modify the specializer on the second `m1` method in Figure 18 to be `ArrayList<String>`. This revised code typechecks in JPred, because Java allows an object of type `List<T>` to be cast to `ArrayList<String>`. Therefore, `l` is considered to have type `ArrayList<String>` when typechecking the body of the second `m1` method. However, at runtime the dynamic dispatch (i.e., the generated `instanceof` test and subsequent cast) succeeds as long as the given list is an `ArrayList`, regardless of the type of its elements. As in Java, the JPred typechecker signals a static warning whenever a program contains an unchecked specializer expression. While it is possible to statically prevent unsound type tests, as others have done [Emir et al. 2007], we have chosen to retain consistency with Java's semantics.

Exhaustiveness and ambiguity checking proceed as described in Section 3.2, except that we always use the erased types for specializers when performing the checking. This ensures, for example, that if a message has two methods, one dispatching on `ArrayList<String>` and the other dispatching on `ArrayList<Integer>`, these methods will be found to be ambiguous. Indeed, due to Java's type erasure, these specializers have the exact same effect dynamically.

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 [Ernst et al. 1998; Ucko 2001; Orleans 2002]. 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, 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 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 CVC3 [2009], which is the automatic theorem prover that our implementation uses. Then we describe how JPred's predicate expressions are represented in CVC3's input language. Finally we describe the axioms we provide to CVC3 so it can reason precisely about objects and classes.

4.1 CVC3

CVC3 is an automatic theorem prover in the Nelson-Oppen style [Nelson and Oppen 1979]. The theorem prover integrates separate decision procedures for several decidable theories, including:

—real and integer linear arithmetic;
—equality with uninterpreted function symbols;

—arrays;
—records; and
—user-defined inductive datatypes.

CVC3's input language allows expression of first-order logic formulas over the aforementioned interpreted theories. While this logic is undecidable in general, we only employ the quantifier-free subset of CVC3's input language, on which CVC3 is sound, complete, and fully automatic. In a typical usage, various formulas are provided as *axioms*, which CVC3 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 CVC3. There are several automatic theorem provers of comparable expressiveness to CVC3 for example, Simplify [Detlefs et al. 2005] and Verifun [Flanagan et al. 2003]. Moving to one of these provers would merely require us to translate the queries and axioms we provide to CVC3 (see the next two subsections) into the input language of the new prover. In fact, the original implementation of the JPred compiler employed CVC3's predecessor, CVC Lite [Barrett and Berezin 2004], which has a slightly different input language.

4.2 Representing Predicate Expressions

Before translating a predicate expression into the syntax of CVC3'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 ith formal name with the name `arg`i everywhere.[3] 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.

It is straightforward to translate predicate expressions in internal form into the syntax of CVC3's input language. All of our allowed unary and binary operators (see Figure 1) are translated to their counterparts in CVC3, as are array accesses and all of the literals except `null`. The literal `null` and all variables and field accesses appearing in a given predicate expression are translated to themselves; they are treated as variable names by CVC3. For example, the target `this.data` is treated as an atomic variable name, with no relation to the target `this`. A pure method call is translated to a call of an associated uninterpreted function symbol, with the receiver argument treated as an ordinary argument to the function. Finally, a specializer expression *tgt*`@`*Type* is translated

[3]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.

as `instanceof`(*tgt*, *Type*), where `instanceof` is a distinguished function symbol that we declare.

For example, consider the `contains` methods in `DataNode` of Figure 15. During static ambiguity checking, the first and second methods are tested for disjointness by posing the following query to CVC3.

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

Here = is CVC3's analog of Java's == operator. CVC3 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 12. After converting their predicates to internal form, we pose the following query to CVC3.

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

Since `instanceof` is an uninterpreted function symbol, CVC3 does not know anything about its semantics. Therefore, CVC3 cannot prove that the preceding formula is valid, even though the two methods are in fact disjoint.

To address this issue, we provide CVC3 with axioms about the semantics of `instanceof`. These axioms effectively mirror the relevant portion of the associated JPred program's subtype relation in CVC3. We call a target a *reference target* if it has reference type. Let F be a query formula provided to CVC3. We consider in turn each pair ($\{T_1, T_2\}$, *tgt*), where T_1 and T_2 are distinct class or interface names mentioned in F and *tgt* is a reference target mentioned in F.

—If T_1 is a subtype of T_2, we declare the axiom `instanceof(`*tgt*`, `T_1`) => instanceof(`*tgt*`, `T_2`)`, where => is the logical implication operator in CVC3.

—Otherwise, if T_2 is a subtype of T_1, we declare the axiom `instanceof(`*tgt*`, `T_2`) => instanceof(`*tgt*`, `T_1`)`.

—Otherwise, if both T_1 and T_2 are classes, we declare the axiom `NOT(instanceof(`*tgt*`, `T_1`) AND instanceof(`*tgt*`, `T_2`))`. This axiom models the fact that classes support only single inheritance.

For the previous `check` example query, we automatically produce the following axiom.

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

In the presence of this axiom, CVC3 can now prove that the aforesaid query is valid, and hence the JPred typechecker will correctly conclude that the first and fourth `check` methods in Figure 12 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 [Ernst et al. 1998].

The axiom-generation scheme is easily augmented to properly handle array types. To complete the semantics of `instanceof`, we also include two other kinds of axioms. First, for each type T in a query formula F, we declare the axiom `NOT(instanceof(null,` T`))`. 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 2.4, 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*`,` T`) OR` *tgt* `= null`, where T 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, a few other kinds of axioms are required to relate objects to their substructure. For example, 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 following 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 14 overrides the second one.

5. IMPLEMENTATION

We have implemented JPred in the Polyglot extensible compiler framework for Java [Nystrom et al. 2003]. The base Polyglot compiler supports only Java 1.4, but our extension for JPred is implemented as an extension to another Polyglot extension that supports Java 1.5 features [Polyglot for Java 5 2009], including generics and the for-each loop.

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 method family: Each method family'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* [Clifton 2001]. 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 CVC3 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.

```
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()); }
```

Fig. 19. The translation of the `isomorphic` methods in Figure 7 to Java.

Our modular compilation strategy generalizes that of MultiJava [Clifton 2001] 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 this 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.

For example, Figure 19 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 method family is completely transparent to whether or not the method family employs predicate dispatch. 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 single most-specific applicable method. If all the methods in d are predicate methods and there exist inherited methods belonging to the same method family, 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 20 contains the dispatcher method for the methods in Figure 19. 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. Similarly, a formal with an escaping specialized type must be cast to the specialized type before

```
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); }
}
```

Fig. 20. A dispatcher method for the methods in Figure 19.

$$
\begin{array}{rcl}
\mathtt{TD} & ::= & \mathtt{class\ C\ extends\ C\ implements\ }\overline{\mathtt{I}}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\} \\
 & | & \mathtt{interface\ I\ extends\ }\overline{\mathtt{I}}\ \{\overline{\mathtt{MH}}\} \\
\mathtt{K} & ::= & \mathtt{C}(\overline{\mathtt{T}}\ \overline{\mathtt{f}})\ \{\mathtt{super}(\overline{\mathtt{f}});\ \mathtt{this}.\overline{\mathtt{f}}\ \mathtt{=}\ \overline{\mathtt{f}};\} \\
\mathtt{M} & ::= & \mathtt{T\ m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\ \mathtt{when}\ \overline{\mathtt{P}}\ \{\mathtt{return}\ \overline{\mathtt{t}};\} \\
\mathtt{MH} & ::= & \mathtt{T\ m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}}); \\
\mathtt{P,Q} & ::= & \mathtt{true} \mid \mathtt{x@T} \mid \neg\mathtt{P} \mid \mathtt{P}\wedge\mathtt{P} \mid \mathtt{P}\vee\mathtt{P} \\
\mathtt{S,T} & ::= & \mathtt{C} \mid \mathtt{I} \\
\mathtt{s,t} & ::= & \mathtt{x} \mid \mathtt{t.f} \mid \mathtt{t.m}(\overline{\mathtt{t}}) \mid \mathtt{new\ C}(\overline{\mathtt{t}}) \mid \mathtt{(T)t} \\
\mathtt{u,v} & ::= & \mathtt{new\ C}(\overline{\mathtt{v}})
\end{array}
$$

Fig. 21. The syntax of FJPred.

invoking a predicate's associated method, as shown in the first branch of the `if` statement. As before, 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. FEATHERWEIGHT JPRED

This section overviews Featherweight JPred (FJPred), an extension of Featherweight Java (FJ) [Igarashi et al. 2001] that formalizes predicate dispatch in JPred. We have formalized the syntax, dynamic semantics, and static semantics of FJPred and have proven a type soundness theorem. We provide the most relevant portions of the formalism here; the full details are available in our companion technical report [Frost and Millstein 2005].

As far as we are aware, FJPred is the first provably sound formalization of predicate dispatch. The concept of multiple dispatch has been formalized in several ways, along with associated type soundness results [Castagna et al. 1995; Castagna 1997; Millstein and Chambers 2002; Millstein et al. 2004]. The original work on predicate dispatch [Ernst et al. 1998] presented a formalization of predicate dispatch but did not prove type soundness.

6.1 Syntax

Figure 21 gives the syntax of FJPred, which augments FJ with interfaces and method predicates. The metavariables `C`, `D`, and `E` range over class names, `I` and `J` over interface names, `f` and `g` over field names, `m` and `n` over method names, and `x` and `y` over parameter names. FJPred has analogous notational conventions and sanity conditions to those in FJ. Like FJ, we use $\overline{\mathtt{D}}$ as a shorthand for a

$\boxed{TT \vdash \texttt{T}_1\texttt{<:}\texttt{T}_2}$

$$TT \vdash \texttt{T<:T}$$

$$\frac{TT \vdash \texttt{T}_1\texttt{<:}\texttt{T}_3 \qquad TT \vdash \texttt{T}_3\texttt{<:}\texttt{T}_2}{TT \vdash \texttt{T}_1\texttt{<:}\texttt{T}_2}$$

$$\frac{TT(\texttt{C}) = \texttt{class C extends D ...}}{TT \vdash \texttt{C<:D}}$$

$$\frac{TT(\texttt{C}) = \texttt{class C extends D implements } \overline{\texttt{I}} \texttt{ ...}}{TT \vdash \texttt{C<:}\texttt{I}_i}$$

$$\frac{TT(\texttt{I}) = \texttt{interface I extends } \overline{\texttt{I}} \texttt{ ...}}{TT \vdash \texttt{I<:}\texttt{I}_i}$$

Fig. 22. The subtyping relation.

sequence of elements from domain D, using the appropriate separator depending on context. Also as in FJ, we abuse this shorthand for convenience when clear from context. For example, $\overline{\texttt{T}}\,\overline{\texttt{x}}$ in the syntax of method declarations is shorthand for a comma-separated sequence of formal parameter declarations. We comment on other conventions throughout as necessary.

For uniformity, all methods have a predicate; a method with the predicate `true` has the same semantics as a regular Java method. Also, the syntax groups all methods of the same method family in each class as a single declaration. In particular, the notation $\texttt{T m(}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{) when }\overline{\texttt{P}}\ \texttt{\{return }\overline{\texttt{t}}\texttt{;\}}$ abbreviates the following method declaration.

$$\texttt{T m(T}_1\ \texttt{x}_1\texttt{, } \cdots\ \texttt{T}_n\ \texttt{x}_n\texttt{) when P}_1\ \texttt{\{return t}_1\texttt{;\}}\ \cdots\ \texttt{when P}_m\ \texttt{\{return t}_m\texttt{;\}}$$

Having all methods for a given method family in one declaration simplifies the formal semantics. However, method lookup is still independent of the textual order; FJPred does not support JPred's ordered dispatch syntactic sugar (but its desugaring is expressible).

Method predicates include type tests on formals and conjunctions, disjunctions, and negations of such tests. We omit the other constructs supported by JPred predicates, as they do not exacerbate the issue of modular typechecking in interesting ways, which is the focus of our formalization.

An FJPred program is a pair of a *type table*, which maps type (class or interface) names to their declarations, and a term. The rules assume a fixed global type table TT, although a few of the judgments additionally include an explicit type table in the context.

6.2 Subtyping

The subtyping relation among types is defined by the rules in Figure 22. The rules extend those of FJ to handle interfaces in a straightforward manner. The rules use an explicit type table in the context, which shadows the implicit global type table; the reason for this will be clear later.

$$\boxed{TT;\Gamma \models \texttt{P}}$$

$$TT;\Gamma \models \texttt{true} \qquad \frac{TT;\Gamma \models \texttt{P}_1 \qquad TT;\Gamma \models \texttt{P}_2}{TT;\Gamma \models \texttt{P}_1 \wedge \texttt{P}_2}$$

$$\frac{\texttt{x:C} \in \Gamma \qquad TT \vdash \texttt{C<:T}}{TT;\Gamma \models \texttt{x@T}} \qquad \frac{TT;\Gamma \models \texttt{P}_1}{TT;\Gamma \models \texttt{P}_1 \vee \texttt{P}_2}$$

$$\frac{TT;\Gamma \not\models \texttt{P}}{TT;\Gamma \models \neg\texttt{P}} \qquad \frac{TT;\Gamma \models \texttt{P}_2}{TT;\Gamma \models \texttt{P}_1 \vee \texttt{P}_2}$$

Fig. 23. Evaluating predicates.

$$\boxed{\overline{\texttt{x}} \models \texttt{P}}$$

$$\frac{\forall\, TT' \supseteq TT.\ \forall\, \overline{\texttt{C}} \subseteq \text{dom}(TT').\, |\overline{\texttt{C}}| = |\overline{\texttt{x}}| \text{ implies } TT';\overline{\texttt{x}}\texttt{:}\overline{\texttt{C}} \models \texttt{P}}{\overline{\texttt{x}} \models \texttt{P}}$$

Fig. 24. Predicate validity.

6.3 Reasoning About Predicates

Figure 23 provides the rules for evaluating predicates; the judgment $TT;\Gamma \models$ P formalizes the conditions under which a predicate evaluates to true. As usual, Γ denotes a type environment, which maps variables to types. Intuitively, Γ provides the runtime classes of the actual arguments to a method, which is necessary to determine if a dynamic dispatch in the method's predicate succeeds.

As described earlier, determining the method overriding relation as well as reasoning about exhaustiveness and ambiguity of method families reduce to checking validity of propositional formulas, and our implementation uses CVC3 for this purpose. It is beyond the scope of this formalization to formally model the particular decision procedures used by CVC3 in order to prove a query valid. Instead, we formalize the *consequence* of a validity query. Figure 24 defines our notion of validity. The judgment $\overline{\texttt{x}} \models$ P indicates that a logical formula P, which uses the same syntax as FJPred predicates, is valid, where $\overline{\texttt{x}}$ are the free variables in P. The associated rule defines a formula to be valid if in all extensions of the current program, for all assignments of classes to the free variables in P, the formula evaluates to true.

The use of all extensions TT' of TT reflects the modularity of validity checking. For example, consider the formula ¬(x@Open ∧ x@SaveAs), where Open and SaveAs are interfaces. Even if a given type table has no class that implements both of these interfaces, our rule ensures that the formula will not be considered valid. Quantifying over all extensions of TT formalizes the conservative nature of the validity check: We must always assume the possibility of a class that implements both Open and SaveAs.[4]

[4]It would be slightly more accurate to quantify over all extensions of a *subset* of TT. Intuitively, this subset includes only the types mentioned in the formula P and their supertypes, as these are the only types that CVC3 is given information about. Our technical report [Frost and Millstein 2005] formalizes this approach, but the two notions of validity can be shown equivalent, so we employ the simpler version here.

$$\boxed{\mathit{mbody}(\mathtt{m},\mathtt{C},\overline{\mathtt{D}}) = (\overline{\mathtt{x}},\mathtt{t})}$$

$$\frac{\begin{array}{c} TT(\mathtt{C}) = \mathtt{class\ C\ extends\ E\ implements}\ \overline{\mathtt{I}}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\} \\ \mathtt{S\ m(}\overline{\mathtt{S}}\ \overline{\mathtt{x}}\mathtt{)\ when}\ \overline{\mathtt{P}}\ \{\mathtt{return}\ \overline{\mathtt{t}};\} \in \overline{\mathtt{M}} \\ TT;\overline{\mathtt{x}}{:}\overline{\mathtt{D}} \models \mathtt{P}_i \qquad \mathit{overridesIfApplicable}(\mathtt{P}_i,\overline{\mathtt{P}},\overline{\mathtt{x}},\overline{\mathtt{D}}) \end{array}}{\mathit{mbody}(\mathtt{m},\mathtt{C},\overline{\mathtt{D}}) = (\overline{\mathtt{x}},\mathtt{t}_i)}$$

$$\frac{\begin{array}{c} TT(\mathtt{C}) = \mathtt{class\ C\ extends\ E\ implements}\ \overline{\mathtt{I}}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\} \\ \mathtt{S\ m(}\overline{\mathtt{S}}\ \overline{\mathtt{x}}\mathtt{)\ when}\ \overline{\mathtt{P}}\ \{\mathtt{return}\ \overline{\mathtt{t}};\} \in \overline{\mathtt{M}} \\ \text{there is no } \mathtt{P}_i \text{ such that } TT;\overline{\mathtt{x}}{:}\overline{\mathtt{D}} \models \mathtt{P}_i \end{array}}{\mathit{mbody}(\mathtt{m},\mathtt{C},\overline{\mathtt{D}}) = \mathit{mbody}(\mathtt{m},\mathtt{E},\overline{\mathtt{D}})}$$

$$\frac{\begin{array}{c} TT(\mathtt{C}) = \mathtt{class\ C\ extends\ E\ implements}\ \overline{\mathtt{I}}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\} \\ \mathtt{m} \text{ is not defined in } \overline{\mathtt{M}} \end{array}}{\mathit{mbody}(\mathtt{m},\mathtt{C},\overline{\mathtt{D}}) = \mathit{mbody}(\mathtt{m},\mathtt{E},\overline{\mathtt{D}})}$$

$$\boxed{\mathit{overridesIfApplicable}(\mathtt{P}_1,\mathtt{P}_2,\overline{\mathtt{x}},\overline{\mathtt{D}})}$$

$$\frac{(\mathtt{P}_1 \not\doteq \mathtt{P}_2 \text{ and } TT;\overline{\mathtt{x}}{:}\overline{\mathtt{D}} \models \mathtt{P}_2) \text{ implies } (\overline{\mathtt{x}} \models \mathtt{P}_1 \Rightarrow \mathtt{P}_2 \text{ and } \overline{\mathtt{x}} \not\models \mathtt{P}_2 \Rightarrow \mathtt{P}_1)}{\mathit{overridesIfApplicable}(\mathtt{P}_1,\mathtt{P}_2,\overline{\mathtt{x}},\overline{\mathtt{D}})}$$

Fig. 25. Method lookup rules.

6.4 Dynamic Semantics

As in FJ, the dynamic semantics of FJPred is formalized with a small-step operational semantics whose main judgment has the form $\mathtt{t}_1 \longrightarrow \mathtt{t}_2$. The majority of the rules are identical to those of FJ except for minor adaptations to account for the presence of interfaces. The only interesting rule is the one for method invocation.

$$\frac{\overline{\mathtt{u}} = \mathtt{new}\ \overline{\mathtt{D}}\mathtt{(\ldots)} \qquad \mathit{mbody}(\mathtt{m},\mathtt{C},\overline{\mathtt{D}}) = (\overline{\mathtt{x}},\mathtt{t}_0)}{\mathtt{(new\ C(}\overline{\mathtt{v}}\mathtt{)).m(}\overline{\mathtt{u}}\mathtt{)} \longrightarrow [\overline{\mathtt{x}} \mapsto \overline{\mathtt{u}},\ \mathtt{this} \mapsto \mathtt{new\ C(}\overline{\mathtt{v}}\mathtt{)}]\mathtt{t}_0}$$

The *mbody* function performs method lookup via predicate dispatch, given the runtime classes of the receiver and the other arguments. The rules defining *mbody* and a helper function are defined in Figure 25. The first rule applies when the receiver class contains a single applicable method that overrides all other applicable methods in that class. The first two premises in the rule identify the method, and the third premise indicates that the method is applicable: Its predicate evaluates to `true` in the context of the given actual argument classes. The final premise uses the *overridesIfApplicable* helper function to check that the method strictly overrides all other applicable methods. The judgment $\mathtt{P}_1 \doteq \mathtt{P}_2$ holds if $\mathtt{P}_1$ and $\mathtt{P}_2$ denote the same textual predicate from the program, and $\mathtt{P}_1 \Rightarrow \mathtt{P}_2$ abbreviates the predicate $\neg\mathtt{P}_1 \vee \mathtt{P}_2$.

The second and third *mbody* rules handle the situation when there are no applicable methods in the receiver class; either there are declared methods for the invoked method family but none are applicable, or there are no declared

$\boxed{\texttt{M OK in C}}$

$$\frac{\overline{\texttt{x}} \vdash \overline{\texttt{P}}\ \texttt{OK} \qquad \overline{\texttt{x}}\texttt{:}\overline{\texttt{T}}\texttt{,this:C} \vdash \overline{\texttt{t}} : \overline{\texttt{S}} \qquad TT \vdash \overline{\texttt{S}}\texttt{<:T} \qquad \forall \texttt{P} \in \overline{\texttt{P}}.\forall \texttt{Q} \in \overline{\texttt{P}}.\mathit{unambiguous}(\texttt{P}, \texttt{Q}, \overline{\texttt{x}}, \overline{\texttt{P}})}{\texttt{T m(}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{) when } \overline{\texttt{P}}\ \{\texttt{return } \overline{\texttt{t}}\texttt{;}\}\ \texttt{OK in C}}$$

$\boxed{\overline{\texttt{x}} \vdash \texttt{P OK}}$

$$\overline{\texttt{x}} \vdash \texttt{true OK}$$

$$\frac{\texttt{x} \in \overline{\texttt{x}}}{\overline{\texttt{x}} \vdash \texttt{x@S OK}} \qquad \frac{\overline{\texttt{x}} \vdash \texttt{P}_1\ \texttt{OK} \qquad \overline{\texttt{x}} \vdash \texttt{P}_2\ \texttt{OK}}{\overline{\texttt{x}} \vdash \texttt{P}_1 \wedge \texttt{P}_2\ \texttt{OK}}$$

$$\frac{\overline{\texttt{x}} \vdash \texttt{P OK}}{\overline{\texttt{x}} \vdash \neg\texttt{P OK}} \qquad \frac{\overline{\texttt{x}} \vdash \texttt{P}_1\ \texttt{OK} \qquad \overline{\texttt{x}} \vdash \texttt{P}_2\ \texttt{OK}}{\overline{\texttt{x}} \vdash \texttt{P}_1 \vee \texttt{P}_2\ \texttt{OK}}$$

$\boxed{\mathit{unambiguous}(\texttt{P}_1, \texttt{P}_2, \overline{\texttt{x}}, \overline{\texttt{P}})}$

$$\frac{\begin{array}{c}\overline{\texttt{x}} \models \texttt{P}_1 \Rightarrow \texttt{P}_2 \text{ and } \overline{\texttt{x}} \models \texttt{P}_2 \Rightarrow \texttt{P}_1 \text{ implies } \texttt{P}_1 \doteq \texttt{P}_2 \\ \overline{\texttt{Q}} = [\texttt{P} \mid \texttt{P} \in \overline{\texttt{P}} \text{ and } \overline{\texttt{x}} \models \texttt{P} \Rightarrow \texttt{P}_1 \text{ and } \overline{\texttt{x}} \models \texttt{P} \Rightarrow \texttt{P}_2] \\ \overline{\texttt{x}} \models (\texttt{P}_1 \wedge \texttt{P}_2) \Rightarrow \bigvee \overline{\texttt{Q}}\end{array}}{\mathit{unambiguous}(\texttt{P}_1, \texttt{P}_2, \overline{\texttt{x}}, \overline{\texttt{P}})}$$

Fig. 26. Typechecking methods.

methods for the given method family. In both cases, lookup proceeds recursively in the direct superclass.

6.5 Static Semantics

As usual, the typechecking rules for expressions are formalized by a judgment of the form $\Gamma \vdash \texttt{t} : \texttt{T}$. Again, the associated rules are straightforward adaptations of the FJ rules. This includes the rule for typechecking method invocations, which is unaffected by whether or not the methods in the invoked method family employ predicate dispatch.

$$\frac{\begin{array}{c}\Gamma \vdash \texttt{t}_0 : \texttt{T}_0 \qquad \mathit{mtype}(\texttt{m}, \texttt{T}_0) = \overline{\texttt{T}} \to \texttt{T} \\ \Gamma \vdash \overline{\texttt{t}} : \overline{\texttt{S}} \qquad TT \vdash \overline{\texttt{S}}\texttt{<:}\overline{\texttt{T}}\end{array}}{\Gamma \vdash \texttt{t}_0\texttt{.m(}\overline{\texttt{t}}\texttt{)} : \texttt{T}}$$

As in FJ, the *mtype* helper function looks up the declared argument and result types of a method `m` in some type `T`, searching the supertypes if no method declaration is found. We augment the function in the obvious way to handle interfaces.

The top rule in Figure 26 defines how methods are typechecked. The first premise typechecks each predicate, using the rules defined in the middle of the figure. These rules simply ensure that the only variables a predicate refers to are the associated method's formals. For simplicity, our formalism does not model typechecks on predicates that are unnecessary for soundness, such as the check for satisfiability. The second and third premises ensure that the method bodies are all type-correct. The body of a method is typechecked in the context of the declared static types of the formals. It would be safe to sometimes narrow these types based on the type tests in the method's predicate. The full JPred language does so, but we have elided it for simplicity.

The final premise performs ambiguity checking on each pair of predicates, as specified informally in Section 3.2.3 and formally in the bottom rule in Figure 26. The rule for ambiguity checking first requires that the two given predicates $\mathtt{P}_1$ and $\mathtt{P}_2$ are not logically equivalent unless they are the same textual predicate. The second premise uses a comprehension notation to collect the subset $\overline{\mathtt{Q}}$ of predicates defined in the current method declaration that override both $\mathtt{P}_1$ and $\mathtt{P}_2$. The final premise then ensures that $\overline{\mathtt{Q}}$ is a resolving set for $\mathtt{P}_1$ and $\mathtt{P}_2$: Whenever both $\mathtt{P}_1$ and $\mathtt{P}_2$ are satisfied, then so is at least one predicate in $\overline{\mathtt{Q}}$. This check subsumes two special cases discussed explicitly in Section 3.2.3. First, if $\mathtt{P}_1$ overrides $\mathtt{P}_2$, then $\mathtt{P}_1$ is in $\overline{\mathtt{Q}}$ so the last premise holds and the methods are considered unambiguous, and similarly for the case when $\mathtt{P}_2$ overrides $\mathtt{P}_1$. Second, if $\mathtt{P}_1$ and $\mathtt{P}_2$ are disjoint, then $\mathtt{P}_1 \wedge \mathtt{P}_2$ is logically false, so the last premise holds vacuously and the methods are considered unambiguous.

Finally, the rules for typechecking classes and interfaces are presented in Figure 27. The first rule in the figure typechecks classes. The first three premises are adapted from FJ, ensuring that the constructor has the appropriate form and typechecking each method declaration; the *fields* helper function obtains a class's fields (including inherited ones) and is defined as in FJ. The *allMethodNames* function returns the names of all methods declared in the given class and in any of its (transitive) supertypes. Each of the associated method families is then checked for proper method overriding and for exhaustiveness.

The rules for method overriding are shown in the middle of Figure 27. Analogous with the rule for method overriding in FJ, we require that a class's supertypes agree with the class on each method's type. We also require a class's supertypes to agree with the class on each method's formal-parameter names, which are accessed by the *mformals* helper function (definition not shown). Requiring agreement on formal-parameter names simplifies exhaustiveness checking by ensuring that all method predicates have the same free variables.

The rule for exhaustiveness checking is shown at the bottom of Figure 27 and formalizes the checking described in Section 3.2.2. The *mpreds* function returns all predicates associated with a method of the given method family in the given class and in all superclasses. The method family is deemed exhaustive for that class if the disjunction of all of these predicates is valid.

6.6 Type Soundness

We have proven a type soundness theorem for FJPred using the standard "progress and preservation" style [Wright and Felleisen 1994]. As in the presentation of FJ, the theorems assume that TD OK holds for each class or interface declaration TD in TT.

THEOREM 6.1 (PROGRESS). *If* $\vdash \mathtt{t} : \mathtt{T}$*, then either* t *is a value,* t *contains a subexpression of the form* (S)(new C($\overline{\mathtt{v}}$)) *where* $TT \vdash \mathtt{C} \not<: \mathtt{S}$*, or there exists some term* s *such that* $\mathtt{t} \longrightarrow \mathtt{s}$.

$\boxed{\texttt{TD OK}}$

$$\frac{\texttt{K} = \texttt{C(}\overline{\texttt{S}}\ \overline{\texttt{g}}\texttt{, }\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{) \{super(}\overline{\texttt{g}}\texttt{); this.}\overline{\texttt{f}}\texttt{ = }\overline{\texttt{f}}\texttt{;\}} \qquad \mathit{fields}(\texttt{D}) = \overline{\texttt{S}}\ \overline{\texttt{g}} \qquad \overline{\texttt{M}}\ \texttt{OK in C} \\ \mathit{allMethodNames}(\texttt{C}) = \overline{\texttt{m}} \qquad \mathit{override}(\overline{\texttt{m}}, \texttt{C}) \qquad \mathit{exhaustive}(\overline{\texttt{m}}, \texttt{C})}{\texttt{class C extends D implements } \overline{\texttt{I}}\ \{\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{; K } \overline{\texttt{M}}\}\texttt{ OK}}$$

$$\frac{\mathit{allMethodNames}(\texttt{I}) = \overline{\texttt{m}} \qquad \mathit{override}(\overline{\texttt{m}}, \texttt{I})}{\texttt{interface I extends } \overline{\texttt{I}}\ \{\overline{\texttt{MH}}\}\texttt{ OK}}$$

$\boxed{\mathit{override}(\texttt{m},\texttt{T})}$

$$\frac{TT(\texttt{C}) = \texttt{class C extends D implements } \overline{\texttt{I}}\ \{\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{; K } \overline{\texttt{M}}\} \\ \mathit{mtype}(\texttt{m}, \texttt{C}) = \overline{\texttt{S}} \rightarrow \texttt{S} \qquad \mathit{mformals}(\texttt{m}, \texttt{C}) = \overline{\texttt{x}} \\ \mathit{override}(\texttt{m}, \texttt{D}, \overline{\texttt{S}} \rightarrow \texttt{S}, \overline{\texttt{x}}) \qquad \mathit{override}(\texttt{m}, \overline{\texttt{I}}, \overline{\texttt{S}} \rightarrow \texttt{S}, \overline{\texttt{x}})}{\mathit{override}(\texttt{m}, \texttt{C})}$$

$$\frac{TT(\texttt{I}) = \texttt{interface I extends } \overline{\texttt{I}}\ \{\overline{\texttt{MH}}\} \\ \mathit{mtype}(\texttt{m}, \texttt{I}) = \overline{\texttt{S}} \rightarrow \texttt{S} \qquad \mathit{mformals}(\texttt{m}, \texttt{I}) = \overline{\texttt{x}} \\ \mathit{override}(\texttt{m}, \overline{\texttt{I}}, \overline{\texttt{S}} \rightarrow \texttt{S}, \overline{\texttt{x}})}{\mathit{override}(\texttt{m}, \texttt{I})}$$

$\boxed{\mathit{override}(\texttt{m}, \texttt{T}, \overline{\texttt{T}} \rightarrow \texttt{T}_0, \overline{\texttt{x}})}$

$$\frac{\mathit{mtype}(\texttt{m}, \texttt{T}) = \overline{\texttt{S}} \rightarrow \texttt{S}_0 \text{ implies } \overline{\texttt{S}} = \overline{\texttt{T}} \text{ and } \texttt{S}_0 = \texttt{T}_0 \\ \mathit{mformals}(\texttt{m}, \texttt{T}) = \overline{\texttt{y}} \text{ implies } \overline{\texttt{y}} = \overline{\texttt{x}}}{\mathit{override}(\texttt{m}, \texttt{T}, \overline{\texttt{T}} \rightarrow \texttt{T}_0, \overline{\texttt{x}})}$$

$\boxed{\mathit{exhaustive}(\texttt{m},\texttt{C})}$

$$\frac{\mathit{mpreds}(\texttt{m}, \texttt{C}) = \overline{\texttt{P}} \qquad \mathit{mformals}(\texttt{m}, \texttt{C}) = \overline{\texttt{x}} \qquad \overline{\texttt{x}} \models \bigvee \overline{\texttt{P}}}{\mathit{exhaustive}(\texttt{m}, \texttt{C})}$$

Fig. 27. Typechecking classes and interfaces.

THEOREM 6.2 (TYPE PRESERVATION). *If* $\Gamma \vdash \texttt{t} : \texttt{T}$ *and* $\texttt{t} \longrightarrow \texttt{s}$*, then there exists some type* $\texttt{S}$ *such that* $\Gamma \vdash \texttt{s} : \texttt{S}$ *and* $TT \vdash \texttt{S<:T}$.

The full proofs of these theorems are available in our companion technical report [Frost and Millstein 2005]. The interesting part of the progress proof involves showing that method lookup always succeeds on well-typed programs, which requires proving the sufficiency of the exhaustiveness and unambiguity typechecks. The type preservation proof is a straightforward generalization of that for FJ.

7. CASE STUDIES

This section describes two case studies that we undertook to evaluate JPred's effectiveness in practice. First, we rewrote portions of one.world, a framework

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

Fig. 28. The interface for event handlers in one.world.

for pervasive computing applications written in Java, to exploit JPred's predicate dispatch. Second, we rewrote the JPred compiler itself to employ JPred's interface dispatch, after this capability was added to the language. A third case study focusing on interface dispatch is described elsewhere [Frost and Millstein 2006].

7.1 Event Handlers in *One.world*

7.1.1 *System Description.* The one.world system is a framework for building pervasive applications in Java, designed and implemented by others [Grimm et al. 2004]. 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 1.4. 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 28: 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 earlier. 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 [Grimm 2002].

7.1.2 *Case Study 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 the most recent version of one.world, version 0.7.1, which is freely available for download [oneworld 2008]. The discovery service implementation consists of two components, `DiscoveryClient` and `DiscoveryServer`, totaling 2120 noncomment, nonblank lines of code (LOC). Together the two components include 11 event handlers as inner classes totaling 876 LOC, or 41% of the code.

```
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;
  }
}
```

Fig. 29. An event handler in `DiscoveryClient`.

Figure 29 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 7.1.3.

Figure 30 illustrates how we implement `MainHandler` from Figure 29 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

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

```
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; }
}
```

Fig. 30. The translation of the code in Figure 29 to JPred.

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 31. 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 is instead declared as one or more `handle1` methods, each declaratively specifying its dispatch constraints. JPred automatically dispatches to the appropriate handler, obviating the need for the original `handle1` method, which performs the dispatch manually. 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 31.

The handlers in Figure 30 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 [Clifton et al. 2006], 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 30 is shown in Figure 32. This style is tedious and forces the dispatching logic to be

```
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; }
  }
}
```

Fig. 31. 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;
}
```

Fig. 32. Simulating field dispatch in MultiJava. `Tuple` is a superclass of `Event` and is the static type of the `tuple` field of `InputResponse`.

spread across multiple method families. Further, as we show in Section 7.1.3, 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

Table I. Quantitative Results

	Java	JPred
methods	22	91
total LOC	765	742
avg LOC	35	8.2
max LOC	175	52
`instanceofs`	80	6
casts	87	22
compile time (sec)	14.1	15.8
theorem prover queries	n/a	557

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 systems other than one.world. The event-based style has been recommended for the implementation of many important classes of applications, from Internet services [Pai et al. 1999; Welsh et al. 2001] to sensor networks [Hill et al. 2000; Gay et al. 2003].

7.1.3 *Results.*

—*Quantitative Results.* We can quantify several properties of the original and rewritten discovery service implementation, which are described in Table I. "Java" refers to the original implementation of the code, and "JPred" refers to the version rewritten in JPred. Twenty-two original methods were rewritten to use JPred's features. These include eleven `handle1` methods and eleven helper methods like `handleBindingResponse`, whose usage is illustrated in Figure 31. These methods have an average size of 35 LOC, with the biggest method being 175 LOC. In the JPred implementation, the elimination of manual dispatching logic causes the number of methods to quadruple and the total code size to be reduced slightly, leading to corresponding reductions in the average and maximum method sizes. The small method sizes indicate that each logical handler is relatively 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 one.world code was written in Java 1.4 and so requires some casts due to the lack of generics.

Table I also includes the time to compile each discovery service implementation. We compiled the Java version with the base Polyglot compiler 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. Both versions require an additional 1.4 seconds to compile the resulting Java source to bytecode using Sun's `javac` compiler. Finally, this case study makes

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

Fig. 33. The need for negation predicates in the case study.

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

Fig. 34. The translation of the code in Figure 33 to JPred.

heavy use of predicate dispatch, requiring 557 queries to the automatic theorem prover.

—*Expressiveness*. Figure 30 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 31 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, in conjunction with ordered dispatch, is also used several times. For example, the event-handling code from `DiscoveryServer` shown in Figure 33 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`, `IN-ACTIVE`, `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 34. Ordered dispatch naturally encodes the dispatch on `state`, and a helper method `handleActive` is used to dispatch on the event (via several predicate methods) when 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 29. After declaring and initializing a variable `entry`, both `entry` and the `AnnounceEvent ae` are dispatched upon, as shown in Figure 35. JPred's

```
...
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 {
    ...
  }
}
```

Fig. 35. Null dispatch and linear arithmetic in the case study.

predicate language is expressive enough to allow this code to be modularized into its conceptual handlers.

Of the 91 methods in the JPred implementation of the discovery service's event handling, only 38 of them either have no predicate or perform MultiJava-style multimethod dispatch. Therefore, MultiJava is unable to express 53 of the methods, or 58%. Of these 53 methods, 44 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 32. However, in many cases, MultiJava would require multiple helper method families to properly simulate a single JPred method, making MultiJava's solution tedious and unnatural. The final 9 methods, or 10% 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 was one event handler where it would have been natural to put two method invocations in the handler's predicate. However, both method invocations employ a `String` literal as an argument, and our implementation does not currently support strings in predicates. Both methods satisfy our definition of purity as described earlier, under the reasonable assumption that the `equals` method of Java's `String` class and the `containsKey` method of Java's `HashMap` class are also pure.

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 35. Because there are several

```
...
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;
```

Fig. 36. An ambiguity found in the discovery service.

statements (represented by the initial ellipsis in the figure) before the dispatch code, a helper method family `handleAnnounceEvent` was created in the JPred implementation, which is invoked after those statements in order to dispatch on `entry` and `ae`.

—*Errors Found.* During the course of the case study, we found ambiguity errors, nonexhaustive errors, and unprotected casts. 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 36. 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.

Seven 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 35, 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` method family. 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 surrounding system; such changes can occur frequently and dynamically in

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

Fig. 37. An unprotected cast found in the discovery service.

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 35 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.

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 37. It is assumed that when the `LeaseEvent`'s `type` is `LeaseEvent.CAN-CELED`, the event's `closure` is a `LocalClosure`. If this 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.

—*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 38.

The original developers could have done away with `handle1`, removed the first three lines from the `handle` method in Figure 38, 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

```
 protected abstract boolean handle1(Event e);

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

Fig. 38. `AbstractHandler`'s `handle` method.

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

Fig. 39. `AbstractHandler`'s `handle` method in JPred.

of the subclass's handlers is applicable, which is tedious and could easily be accidentally omitted. Even with the `handle1` helper method family, subclasses must still explicitly return false to invoke the inherited error handler.

In contrast, JPred can support the desired handler inheritance in the natural way, as shown in Figure 39. The `handle1` method family is no longer necessary. `AbstractHandler` provides two handlers, one for `ExceptionalEvents` 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.

7.2 Interface Dispatch in the JPred Compiler

As mentioned earlier, the JPred compiler is written as an extension in Polyglot [Nystrom et al. 2003]. There are several natural opportunities for employing predicate dispatch in the implementation of a Polyglot extension itself, most notably in the code for a new compiler pass. Polyglot supports the easy addition of new visitors [Gamma et al. 1995] over the abstract syntax tree (AST) nodes. The visitors that come with Polyglot often must employ `instanceof` tests and type casts in order to provide specialized behavior for each kind of AST node. Therefore, we implemented the visitors in the JPred compiler in JPred, allowing the dispatch constraints to be declaratively specified and statically checked for exhaustiveness and unambiguity, similar to the style illustrated by our hypothetical `TypeCheck` visitor in Figure 5.

```
public class DispatcherBuilder extends ContextVisitor {
  protected Node leaveCall(Node n)
    when n@ClassBody_c { ... }
  }
```

Fig. 40. A simple usage of class dispatch in the JPred compiler.

```
public class DispatcherBuilder extends ContextVisitor {
  protected Node leaveCall(Node n)
    when n@ClassBody { ... }
}
```

Fig. 41. Interface dispatch version of Figure 40.

When we first implemented these visitors, JPred did not support interface dispatch. Therefore, the only way to obtain the benefits of predicate dispatch was to dispatch directly on AST node implementation classes. However, Polyglot uses a hierarchy of Java *interfaces* to represent the various AST nodes, with the parallel hierarchy of implementation classes meant to be hidden from clients. This strict separation of interface and implementation is critical to a number of Polyglot's extensibility mechanisms. By violating the intended Polyglot style, our visitors were extremely brittle in the face of later evolution or extension to the compiler.

After we added support for modularly typesafe interface dispatch in the JPred compiler, as described in Section 2.2, we were able to rewrite the compiler itself to exclusively employ interface dispatch instead of class dispatch for the purposes of dispatching on AST nodes. In total, there were 28 method families whose method implementations were converted from using class dispatch to using interface dispatch. In 14 of these cases, the method family contained only a single predicate method (in addition to one or more methods without a predicate). For the most part, converting these cases was as simple as replacing each textual class dispatch in the predicate by the corresponding interface dispatch; Polyglot's naming convention is that `N_c` is the name of the node class implementing interface `N`. For example, Figure 40 shows some code using class dispatch, and Figure 41 shows the version modified to employ interface dispatch.

The other 14 method families we modified each contained between 2 and 12 predicate methods, with a median of 5. To handle these method families, we converted class dispatches to interface dispatches as shown before, and we additionally used the ordered dispatch syntactic sugar to allow modular ambiguity checking to succeed. Figure 42 shows a simple example involving two predicate methods, and Figure 43 shows the version modified to employ interface dispatch.

In Figure 43, the use of ordered dispatch resolves the potential ambiguity between the first two methods: The first method will be invoked if an instance of a class implementing both `Unary` and `Binary` is ever passed to `checkLinearity`. However, in this case we are simply assuming that such a scenario cannot occur, since it does not make sense for an AST node to represent both a unary and a binary expression. Indeed, this scenario would likely be indicative of a

```
public static void checkLinearity(Expr e)
  when e@Unary_c { ... }
public static void checkLinearity(Expr e)
  when e@Binary_c { ... }
public static void checkLinearity(Expr e) { ... }
```

Fig. 42. An example with multiple predicate methods.

```
public static void checkLinearity(Expr e)
  when e@Unary { ... }
| when e@Binary { ... }
| { ... }
```

Fig. 43. Interface dispatch version of Figure 42.

program error. If desired, the programmer can catch such errors at runtime by adding a new method with predicate `e@Unary && e@Binary` this appropriately handles the erroneous scenario. However, this approach becomes prohibitively burdensome as the number of interfaces dispatched upon increases. This limitation is not unique to JPred. For example, manual dispatch in Java using an `if` statement that performs a linear sequence of `instanceof` tests suffers from the same problem.

When converting a set of predicate methods to use ordered dispatch, care must be taken to ensure that the previously unordered methods are placed in the appropriate textual order. The compile-time check for unsatisfiable method predicates described in Section 3 turned out to be a useful sanity check for proper textual ordering. With this check, the JPred compiler was able to debug itself! In particular, running the JPred compiler on itself caused the unsatisfiability check to fail for an ordered dispatch declaration in which a predicate of the form `p@PredicateSpecial` was being tested after a predicate of the form `p@PredicateTarget`, where `PredicateSpecial` is a subinterface of `PredicateTarget`. It was easy to miss this error by manual inspection because the erroneous ordered dispatch declaration consisted of nine cases, of which the two cases causing the error were textually the third and eighth ones.

Finally, Table II contains the quantitative results comparing the compilation of the original version of JPred, compiled with itself, and the modified version of JPred containing only interface dispatch, compiled with itself. The possibility of multiple inheritance requires somewhat more queries to the automatic theorem prover in order to ensure unambiguity, causing a small increase in overall compile time.

8. RELATED WORK

There have been several previous languages containing a form of predicate dispatch. The original work by Ernst et al. [1998] was discussed in the Introduction and throughout the article. They implemented predicate dispatch in an interpreter for Dubious [Millstein and Chambers 2002], a simple core language for formal study of multimethod-based languages. That implementation did not include their static type system.

Table II. Quantitative Results

	Compile Time (secs)	Theorem Prover Queries
JPred-orig	45.9	217
JPred-interface	47.3	310

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 substantially 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. Also, as mentioned in Section 3, the static type system described by Ernst et al. is global, while ours retains Java's modular typechecking strategy. Finally, Ernst et al. formalized their semantics for predicate dispatch but did not prove a soundness result.

Ucko [2001] describes an extension of the Common Lisp Object System (CLOS) [Steele Jr. 1990; Gabriel et al. 1991] 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 [Orleans 2002] is a language that unifies predicate dispatch with features of aspect-oriented programming (AOP) [Kiczales et al. 1997]. 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 nonaround methods, thereby supporting the addition of new cross-cutting code. The language is implemented as a library extension to Scheme [Abelson et al. 1998]. Similar to the two languages described earlier, 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 [1999] 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., tests 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.

A predicate class [Chambers 1993] is a precursor to predicate dispatch that allows an instance of a class C to be considered 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 [Hosking et al. 1990] provide similar capabilities to predicate classes but they use the textual order of methods to define the overriding relation, as in functional languages.

Several OO languages, including Objective Caml (OCaml) [Rémy and Vouillon 1998], JMatch [Liu and Myers 2003], and Scala [Emir et al. 2007], include a form of pattern matching inspired by functional languages. Pattern matching in these languages is independent of the language's OO dynamic dispatch mechanism. Further, the set of cases in a pattern-matching expression is not extensible, and the textual order of cases defines the overriding relation. An exception is OOMatch [Richard and Lhotak 2008], an extension of Java that integrates pattern matching with ordinary dynamic dispatch in a manner similar to JPred's predicate methods.

The patterns supported by pattern matching in these languages are less expressive than JPred's predicate language. However, each language has something to offer that is missing in JPred. Pattern matching in OCaml is a first-class expression and need not appear at the top level of a function's implementation. JMatch's patterns support a sophisticated form of bidirectional computation, as in logic programming languages, which enables idioms like declarative iteration over a collection. Scala and OOMatch respectively provide *extractors* and *deconstructors*, which are programmer-defined patterns for a class. These constructs enable pattern matching on objects of a class without requiring the class's underlying representation to be exposed to clients.

Several languages, including XStatic [Gapeyev and Pierce 2003], CDuce [Benzaken et al. 2003], and HydroJ [Lee et al. 2003], support pattern matching for XML-like [Bray et al. 2006] 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 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 [Millstein et al. 2002] to support modular typechecking of patterns.

Our previous languages MultiJava [Clifton et al. 2006] and Extensible ML (EML) [Millstein et al. 2004] 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 on classes and equality tests against constants, and only for formal parameters. MultiJava also supports *open classes*, that is, 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 dispatch on interfaces and array types, array access expressions, pure methods, disjunction and negation, 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.

9. 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 formalized a core subset of JPred as an extension to Featherweight Java and proved a type soundness theorem. We also performed case studies that demonstrate the practical utility of JPred on existing Java applications, 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 useful to support the other Java literals along with their associated operations. The ability to define Ernst-style predicate abstractions [Ernst et al. 1998] would be convenient and would not cause any technical problems. Finally, support of Scala-style extractors [Emir et al. 2007] would make predicate dispatch more expressive in the face of class encapsulation.

REFERENCES

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

Agrawal, R., DeMichiel, L. G., and Lindsay, B. G. 1991. Static type checking of multi-methods. In *Proceedings of the (OOPSLA'91) Conference on Object-Oriented Programming Systems, Languages and Applications*, 113–128.

Arnold, K., Gosling, J., and Holmes, D. 2005. *The Java Programming Language*, 4th ed. Prentice Hall.

Assaad, M. G. and Leavens, G. T. 2001. Alias-Free parameters in C for better reasoning and optimization. Tech. Rep. 01-11, Department of Computer Science, Iowa State University, Ames, Iowa. November.

Barrett, C. and Berezin, S. 2004. CVC Lite: A new implementation of the cooperating validity checker. In *Proceedings of the* 16th *International Conference on Computer Aided Verification (CAV'04)*, R. Alur and D. A. Peled, Eds. Lecture Notes in Computer Science, vol. 3114. Springer, 515–518.

Baumgartner, G., Jansche, M., and Laufer, K. revised March 2002. Half & Half: Multiple dispatch and retroactive abstraction for Java. Tech. rep. OSU-CISRC-5/01-TR08, Department of Computer and Information Science, The Ohio State University.

Benzaken, V., Castagna, G., and Frisch, A. 2003. CDuce: An XML-centric general-purpose language. In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming*. ACM Press, 51–63.

Bobrow, D. G., Kahn, K., Kiczales, G., Masinter, L., Stefik, M., and Zdybel, F. 1986. CommonLoops: Merging Lisp and object-oriented programming. In *Proceedings of the Conference on Object-oriented Programming Systems, Languages and Applications*. ACM Press, 17–29.

Boyland, J. and Castagna, G. 1997. Parasitic methods: Implementation of multi-methods for Java. In *Proceedings of the OOPSLA'97. ACM SIGPLAN Not. 32*, 10, 66–76.

Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F. 2006. eXtensible markup language (XML) 1.0 (4th Edition). World Wide Web Consortium recommendation, `http://www.w3.org/TR/REC-xml`.

Castagna, G. 1995. Covariance and contravariance: Conflict without a cause. *ACM Trans. Program. Lang. Syst. 17,* 3, 431–447.

Castagna, G. 1997. *Object-Oriented Programming: A Unified Foundation*. Progress in Theoretical Computer Science. Birkhauser, Boston.

Castagna, G., Ghelli, G., and Longo, G. 1995. A calculus for overloaded functions with subtyping. *Inf. Comput. 117,* 1, 115–135.

Chambers, C. 1992. Object-Oriented multi-methods in Cecil. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'92)*, O. L. Madsen, Ed. LNCS 615. Springer-Verlag, 33–56.

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

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

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

Clifton, C. 2001. MultiJava: Design, implementation, and evaluation of a Java-compatible language supporting modular open classes and symmetric multiple dispatch. Tech. rep. 01-10, Department of Computer Science, Iowa State University, Ames, Iowa. November.

Clifton, C., Leavens, G. T., Chambers, C., and Millstein, T. 2000. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *Proceedings of the Conference on Object-Oriented*

Programming, Systems, Languages, and Applications (OOPSLA'00). ACM SIGPLAN Notices, 35, 10, 130–145.

Clifton, C., Millstein, T., Leavens, G. T., and Chambers, C. 2006. Multijava: Design rationale, compiler implementation, and applications. *ACM Trans. Program. Lang. Syst. 28,* 3, 517–575.

CVC3. 2009. The CVC3 home page. `http://www.cs.nyu.edu/acsys/cvc3`.

Detlefs, D., Nelson, G., and Saxe, J. B. 2005. Simplify: A theorem prover for program checking. *J. ACM 52,* 3, 365–473.

Eclipse. 2007. Eclipse home page. `http://www.eclipse.org`.

Emir, B., Odersky, M., and Williams, J. 2007. Matching objects with patterns. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, E. Ernst, Ed. Lecture Notes in Computer Science, vol. 4609. Springer, 273–298.

Ernst, M., Kaplan, C., and Chambers, C. 1998. Predicate dispatching: A unified theory of dispatch. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'98)*, E. Jul, Ed. Lecture Notes in Computer Science, vol. 1445. Springer, 186–211.

Flanagan, C., Joshi, R., Ou, X., and Saxe, J. B. 2003. Theorem proving using lazy proof explication. In *the International Conference on Computer Aided Verification (CAV)*.

Frost, C. and Millstein, T. 2005. Featherweight JPred. Tech. rep. CSD-TR-050038, UCLA Computer Science Department. `ftp://ftp.cs.ucla.edu/tech-report/2005-reports/050038.pdf`.

Frost, C. and Millstein, T. 2006. Modularly typesafe interface dispatch in JPred. In *the International Workshop on Foundations and Developments of Object-Oriented Languages(FOOL/WOOD)*.

Gabriel, R. P., White, J. L., and Bobrow, D. G. 1991. CLOS: Integrating object-oriented and functional programming. *Commun. ACM 34,* 9, 28–38.

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

Gapeyev, V. and Pierce, B. C. 2003. Regular object types. In *Proceedings of the European Conference on Object-Oriented Programming*. Lecture Notes in Computer Science, vol. 2743. Springer.

Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., and Culler, D. 2003. The NesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 1–11.

Gosling, J., Joy, B., Steele, G., and Bracha, G. 2005. *The Java Language Specification 3rd Ed.* Prentice Hall.

Grimm, R. 2002. System support for pervasive applications. Ph.D. thesis, Department of Computer Science and Engineering, University of Washington.

Grimm, R., Davis, J., Lemar, E., Macbeth, A., Swanson, S., Anderson, T., Bershad, B., Borriello, G., Gribble, S., and Wetherall, D. 2004. System support for pervasive applications. *ACM Trans. Comput. Syst. 22,* 4, 421–486.

Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D. E., and Pister, K. S. J. 2000. System architecture directions for networked sensors. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'00)*. 93–104.

Hosking, J. G., Hamer, J., and Mugridge, W. 1990. Integrating functional and object-oriented programming. In *Proceedings of the International Conference on Technology of Obeject-Oriented Languages and Systems TOOLS Pacific*, 345–355.

Igarashi, A., Pierce, B. C., and Wadler, P. 2001. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program, Lang. Syst. 23,* 3, 396–450.

Ingalls, D. H. H. 1986. A simple technique for handling multiple polymorphism. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*. ACM Press, 347–349.

Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. 1997. Aspect-Oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented (ECOOP'97)*, M. Akşit and S. Matsuoka, Eds. Lecture Notes on Computer Science, vol. 1241. Springer, 220–242.

Leavens, G. T. and Antropova, O. 1999. ACL — Eliminating parameter aliasing with dynamic dispatch. Tech. rep. 98-08a, Department of Computer Science, Iowa State University, Ames, Iowa. February.

LEE, K., LAMARCA, A., AND CHAMBERS, C. 2003. HydroJ: Object-Oriented pattern matching for evolvable distributed systems. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*.

LIU, J. AND MYERS, A. C. 2003. JMatch: Iterable abstract pattern matching for Java. In *Proceedings of the International Symposium on Practical Aspects of Declarative Languages (PADL)*, V. Dahl and P. Wadler, Eds. Lecture Notes in Computer Science, vol. 2562. Springer, 110–127.

MILLSTEIN, T. 2003. Reconciling software extensibility with modular program reasoning. Ph.D. thesis, Department of Computer Science and Engineering, University of Washington.

MILLSTEIN, T. 2004. Practical predicate dispatch. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04)*.

MILLSTEIN, T., BLECKNER, C., AND CHAMBERS, C. 2002. Modular typechecking for hierarchically extensible datatypes and functions. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'02). ACM SIGPLAN Not. 37*, 9, 110–122.

MILLSTEIN, T., BLECKNER, C., AND CHAMBERS, C. 2004. Modular typechecking for hierarchically extensible datatypes and functions. *ACM Trans. Program. Lang. Syst. 26,* 5, 836–889.

MILLSTEIN, T. AND CHAMBERS, C. 2002. Modular statically typed multimethods. *Inf. Comput. 175,* 1, 76–118.

MILNER, R., TOFTE, M., HARPER, R., AND MACQUEEN, D. 1997. *The Definition of Standard ML (Revised)*. The MIT Press.

MOON, D. A. 1986. Object-Oriented programming with Flavors. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*. ACM Press, 1–8.

NELSON, G. AND OPPEN, D. C. 1979. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst. 1,* 2, 245–257.

NYSTROM, N., CLARKSON, M. R., AND MYERS, A. C. 2003. Polyglot: An extensible compiler framework for Java. In *Proceedings of the 12th International Conference on Compiler Construction (CC)*. Springer.

oneworld. 2008. one.world home page. `http://cs.nyu.edu/rgrimm/one.world`.

ORLEANS, D. 2002. Incremental programming with extensible decisions. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*. ACM Press, 56–64.

PAI, V. S., DRUSCHEL, P., AND ZWAENEPOEL, W. 1999. Flash: An efficient and portable Web server. In *Proceedings of the USENIX Annual Technical Conference (USENIX-99)*. USENIX Association, 199–212.

Polyglot for Java 5. Polyglot for Java 5 homepage. `http://www.cs.ucla.edu/~milanst/projects/polyglot5`.

RÉMY, D. AND VOUILLON, J. 1998. Objective ML: An effective object-oriented extension of ML. *Theory Practice Object Syst. 4,* 1, 27–52.

RICHARD, A. AND LHOTAK, O. 2008. OOMatch: Pattern matching as dispatch in Java. In *International Workshop on Foundations of Object-Oriented Languages*.

SALCIANU, A. AND RINARD, M. 2005. Purity and side effect analysis for Java programs. In *Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation*.

STEELE JR., G. L. 1990. *Common Lisp: The Language, 2nd Ed.* Digital Press, Bedford, MA.

STROUSTRUP, B. 1997. *The C++ Programming Language: Third Edition*. Addison-Wesley, Reading, MA.

UCKO, A. M. 2001. Predicate dispatching in the common lisp object system. Tech. rep. 2001-006, MIT Artificial Intelligence Laboratory.

WELSH, M., CULLER, D., AND BREWER, E. 2001. SEDA: An Architecture for well-conditioned, scalable internet services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, G. Ganger, Ed. 230–243.

WRIGHT, A. K. AND FELLEISEN, M. 1994. A syntactic approach to type soundness. *Inf. Comput. 115,* 1, 38–94.

Received April 2007; revised January 2008; accepted April 2008