Modularly Typesafe Interface Dispatch in JPred

Christopher Frost Todd Millstein
University of California, Los Angeles
{frost,todd}@cs.ucla.edu

Abstract

Multiple dispatch generalizes the receiver-oriented dynamic dispatch of traditional object-oriented (OO) languages by allowing the run-time classes of all arguments to be employed. While research over the last decade has shown how to integrate multiple dispatch with the modular static typechecking found in traditional OO languages, that work has been forced to impose unnatural restrictions or modifications in order to safely accommodate multiple inheritance. In the context of Java, the effect has been to make it difficult to dispatch on interfaces.

In this paper, we illustrate how the concept of *predicate dispatch*, which generalizes multiple dispatch by allowing each method to be guarded by a predicate indicating when the method should be invoked, provides a simple but practical way to support dispatch on interfaces while preserving modular typechecking. We have instantiated our approach in the context of JPred, an existing extension to Java supporting predicate dispatch that previously disallowed dispatch on interfaces altogether. We have formalized our approach in a core subset of JPred and proven an associated type soundness theorem. We have also performed two case studies using JPred, on the JPred compiler itself and on portions of Eclipse, to demonstrate the utility of our approach in practice.

1. Introduction

Multiple dispatch [33, 5] is a natural generalization of the form of dynamic dispatch found in traditional object-oriented (OO) languages like Smalltalk [23] and Java [2]. With multiple dispatch, the method to be invoked upon a message send is determined based on the dynamic classes of any subset of the message's arguments, rather than just the distinguished *receiver* argument. Multiple dispatch can be naturally applied to the implementation of several common programming idioms [27], including binary methods [7], event-driven systems, and the visitor design pattern [22].

While the concept of multiple dispatch originated a dynamically typed setting, later research has reconciled multiple dispatch with the modular static typechecking found in today's mainstream OO languages like Java. However, to achieve modular typechecking, all of the proposed approaches place severe restrictions or unnatural modifications on the ways in which multiple dispatch interacts with multiple inheritance [1, 6, 16, 3, 30]. In the context of Java, these restrictions and modifications affect the ability to dynamically dispatch on interfaces, which we term *interface dispatch*, greatly hindering that idiom despite its practical utility.

In this paper, we describe a simple but practical way to integrate multiple dispatch with multiple inheritance while preserving modular typechecking. Our key observation is that the notion of *predicate dispatch* [19], which generalizes multiple dispatch by guarding each method with a predicate indicating when the method should be invoked, naturally allows potential multiple-inheritance ambiguities to be modularly resolved without requiring extra restrictions or modifications. We have implemented our approach in the context of JPred [28], an existing extension to Java supporting predicate dispatch. JPred originally disallowed interface dispatch altogether because of the problems for modular typechecking, but our approach allows JPred to now support unrestricted usage of interface dispatch.

We have validated our approach in two ways. First, we have formalized JPred's interface dispatch in an extension of Featherweight Java [25] that we call Featherweight JPred (FJPred), and we have proven the associated modular type system sound. FJPred is of independent interest, since it is the first provably sound formalization of predicate dispatch of which we are aware. Second, we have undertaken two case studies using JPred. We modified the JPred compiler, which is written in JPred on top of the Polyglot extensible compiler framework [36], to use interface dispatch instead of class dispatch. We also rewrote portions of Eclipse [18] to use JPred, relying heavily on interface dispatch. The case studies illustrate the practical utility of our approach to interface dispatch, including its use in the detection of several errors.

The next section presents the necessary background information about multiple dispatch and the problem of combining modular typechecking with multiple inheritance, using the MultiJava [16] extension to Java as an example. Section 3 reviews JPred and describes our modular type system that supports interface dispatch in a practical way. Section 4 presents the FJPred formalism, and Section 5 discusses our experience using JPred's approach to interface dispatch in the case studies.

2. Background

2.1 Multiple Dispatch

Figure 1 shows a simple example of multiple dispatch in MultiJava [16]. Multiple dispatch is used to add an optimization pass to a hypothetical compiler, without having to modify the original classes representing the abstract syntax tree (AST) nodes. The first two methods in `Optimizer` are *multimethods*, dynamically dispatching on the node argument to `optimize` in addition to the implicit receiver argument. For example, the `@If` annotation on the first `optimize` method indicates that this method can only be invoked upon a `optimize` message send if the run-time class of the actual argument for `n` is an instance of `If` or a subclass. Since `If` and `While` are both subclasses of `Node`, each of the first

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright © ACM 0-89791-88-6/97/05...$5.00.

```
abstract class Node { ... }
class If extends Node { ... }
class While extends Node { ... }

class Optimizer {
  Node optimize(Node@If n) { ... }
  Node optimize(Node@While n) { ... }
  Node optimize(Node n) { return n; }
}
```

Figure 1. Multiple dispatch in MultiJava.

two `optimize` methods in Figure 1 overrides the last `optimize` method, which can be viewed as implicitly dynamically dispatching on the `Node` class itself.

When a message is sent in MultiJava, the unique applicable method that overrides all other applicable methods is dynamically selected and invoked; the textual order of the methods is irrelevant. For example, consider the message send `new Optimizer().optimize(new If(...))`. The first and third `optimize` methods are applicable to this message send, while the second method is not applicable since `If` is not a subclass of `While`. Of the two applicable methods, the `@If` method is chosen because it overrides the other applicable method. If a message send has no applicable methods, a *message-not-understood* error is signaled at run time. If a message send has at least one applicable method but no unique applicable method that overrides all others, a *message-ambiguous* error is signaled at run time.

A precise description of MultiJava's method-invocation semantics is available elsewhere [16, 15]. MultiJava employs the *symmetric* semantics of multimethod dispatch, which is used by several other languages as well [11, 10, 14, 39, 16, 30]. The other common approach is the *encapsulated* semantics of multimethod dispatch, which is also used by many languages [34, 8, 7, 6, 3]. These approaches only differ in when a method in some class *C* is considered to override a method declared in a superclass of *C*. This distinction is orthogonal to the problem described in this paper, which affects both approaches equally, and our proposed solution can be easily applied to languages using either semantics.

2.2 Modular Typechecking and Multiple Inheritance

Static typechecking ensures, among other things, that a *message-not-understood* or *message-ambiguous* error cannot occur at run time. Conceptually, this involves checking that for each message send, method lookup succeeds for each type-correct tuple of argument classes to that message. For example, consider checking the `optimize` message from Figure 1. The first `optimize` method will be invoked upon a message send if the given AST node is an instance of `If` or a subclass, the second method will be invoked if the node is an instance of `While` or a subclass, and the third method will be invoked otherwise.

As described, the above typecheck is *global*, as it requires knowledge of all type-correct argument classes for each message. In the context of a modular typechecking regime, as in a language like Java, this typecheck must be performed given only the set of classes that are modularly available. In our example, it may be the case when typechecking `optimize` that some subclasses of `Node` are not available, and indeed it would be unsafe to assume that all such subclasses are known.

Several conservative modular type systems have been proposed for multiple dispatch [8, 6, 30, 29], and these type systems are able to successfully typecheck the `optimize` message. For example, consider the modular type system underlying MultiJava and refined by later languages [29, 28]. To prevent run-time *message-not-understood* errors, the typechecker checks that `optimize` is *exhaustive*, meaning that it has at least one applicable method for every possible type-correct argument tuple. Exhaustiveness is assured by the third `optimize` method in Figure 1, which can handle an arbitrary `Node` subclass. To prevent run-time *message-ambiguous* errors, the typechecker checks that each pair of `optimize` methods is *unambiguous*. In `Optimizer`, the first and third methods are unambiguous because the first method strictly overrides the third one, and similarly for the second and third methods. Finally, the first and second methods are unambiguous because they are *disjoint*, meaning that they cannot be simultaneously applicable.

```
interface Node { ... }
interface If extends Node { ... }
interface While extends Node { ... }
```

Figure 2. A revised node hierarchy.

Unfortunately, the fact that `optimize` 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 `optimize` methods are disjoint, since the language prevents the existence of a class that subclasses both `If` and `While`. Consider instead a variant of our AST node hierarchy in which Java interfaces are used instead of classes, as shown in Figure 2. With this new hierarchy, the first and second `optimize` methods may no longer be disjoint, since there could exist a class that implements both the `If` and `While` interfaces. If an instance of such a class were ever passed to `optimize`, a *message-ambiguous* error would occur at run time.

Given whole-program knowledge, the typechecker would only need to reject the given `optimize` methods as ambiguous if there actually existed a class `C` implementing both the `If` and `While` interfaces. Further, in that case, the programmer could resolve the ambiguity and satisfy the typechecker by adding an `optimize` method of the form

```
Node optimize(Node@C n) { ... }
```

for each such class `C`. However, any sound modular type system must always conservatively reject the `optimize` methods in Figure 1 (in the context of the node hierarchy of Figure 2), since there could exist a multiply inheriting class `C` that is not modularly visible. Further, there is no way for the programmer to resolve this potential ambiguity.

Therefore, the potential for multiple-inheritance ambiguities that are not modularly detectable makes interface dispatch almost impossible to employ in practice. Unfortunately, interface dispatch is often extremely useful. For example, it is common for a framework to expose only interfaces to clients, keeping the underlying implementation classes hidden. This is the case in both Polyglot [36] and Eclipse [18]. Without interface dispatch, clients of these frameworks cannot enjoy the benefits of multiple dispatch.

2.3 Prior Approaches

There have been several prior approaches to handling the ambiguities that arise from the interaction of multiple dispatch and multiple inheritance. All of these approaches either forgo modular ambiguity checking, impose severe restrictions on the ways in which multiple dispatch or multiple inheritance may be employed, or impose unnatural modifications on the semantics of multiple dispatch or multiple inheritance.

The simplest approach is to perform no compile-time ambiguity checking whatsoever, thereby admitting the possibility of *message-ambiguous* errors at run time. This approach is taken by the dynamically typed languages CommonLoops [5], CLOS [41, 38], and Dylan [20, 39], as well as by Cmm [40], a multimethod extension to C++ [42]. Another common approach is to perform static ambiguity checking but to require the whole program to be available in order for such checking to be sound. This approach is taken by the languages Cecil [11, 12], Tuple [26], the C++ extension `doublecpp` [4], and the Java extension Nice [35].

MultiJava [16] safely performs modular ambiguity checking and avoids the problem of multiple inheritance simply by forbidding Java interfaces from being dynamically dispatched upon: the `@T` syntax requires `T` to be a class rather than an interface. Relaxed MultiJava [31] extends MultiJava to allow interface dispatch among other idioms, but it requires load-time checks on classes to ensure that no multiple-inheritance ambiguities arise.

Dubious [30] is a prototype-based OO calculus containing both multimethods and objects that support multiple inheritance. Dubious has a modular type system called *System M*, where a Dubious *module* is the unit of typechecking. To make typechecking sound, System M requires that only single inheritance is used across module boundaries; arbitrary multiple inheritance is allowed within a module. This restriction ensures that if C inherits from both B and A, then C is modularly available whenever either B or A is available, so all potential multimethod ambiguities caused by B and A can be modularly detected.

Half & Half [3] is an extension to Java supporting a form of multimethods, among other things. Half & Half allows interfaces to be dynamically dispatched upon. To retain modular type safety, Half & Half requires that if a message has methods that dispatch on two unrelated interfaces, then at least one of the interfaces and all of its subinterfaces must not be declared `public`. This restriction ensures that any class that causes a multimethod ambiguity for some message will be defined in the same package as the class containing the potentially ambiguous methods. Therefore, a check of all classes declared in that package is sufficient to detect any ambiguities.

A final way to preserve modular typechecking is to modify the semantics of either multiple inheritance or multiple dispatch in order to "define away" ambiguities. The database programming language Polyglot [1] contains CLOS-style multimethods. Ambiguities cannot arise due to multiple inheritance, because Polyglot linearizes the specificity of each class's superclasses, effectively reducing multiple inheritance to single inheritance. Boyland and Castagna [6] describe an extension of Java supporting *parasitic methods*, which provide a form of multiple dispatch on interfaces. The multimethod lookup semantics is similar to that described for MultiJava in Section 2.1. However, in the case of an ambiguity, the textually last ambiguous method is considered to override the others.

3. Interface Dispatch in JPred

3.1 Predicate Dispatch

Predicate dispatch [19] allows each method to be guarded by a predicate indicating when the method is applicable. By including a form of run-time type test in the predicate language, predicate dispatch subsumes the expressiveness of multiple dispatch. The overriding relation on methods also naturally generalizes that used in multiple dispatch: method m_1 is considered to override method m_2 if m_1's predicate logically implies m_2's predicate.

In previous work, we designed and implemented JPred [28], an extension to Java supporting predicate dispatch. In addition to dynamic dispatch on formal parameters, JPred's predicate language supports dispatch on in-scope fields, linear arithmetic predicates, identifier binding, and arbitrary conjunctions, disjunctions, and negations of such predicates. JPred adapts and generalizes the ideas underlying MultiJava to support modular typechecking of predicate dispatch. Prior languages supporting predicate dispatch either do not support static exhaustiveness and ambiguity checking [43, 37] or require the whole program to be available in order for such checking to be sound [19].

```
class Optimizer {
  Node optimize(Node n) when n@If { ... }
  Node optimize(Node n) when n@While { ... }
  Node optimize(Node n) { return n; }
}
```

Figure 3. The optimizer in JPred.

Figure 3 shows a JPred implementation of the compiler optimizer from Figure 1. Methods now have an optional `when` clause containing the associated predicate; a method without a `when` clause is equivalent to one whose `when` clause has the predicate `true`. We refer to a method that has a `when` clause as a *predicate method*. While the three `optimize` methods shown are equivalent to the three MultiJava methods in Figure 1, predicate dispatch allows additional expressiveness. For example, a programmer could employ dispatch on fields, along with the ability to conjoin predicates, to implement a form of constant folding for addition in our optimizer:

```
Node optimize(Node n)
  when n@Plus && n.left@IntLit && n.right@IntLit
  { return new IntLit(n.left.val + n.right.val); }
```

JPred is currently designed and implemented as an extension to Java 1.4. Moving to Java 1.5 would require support for parametric polymorphism. We have previously formalized a variant of MultiJava's modular type system in the presence of parametric polymorphism and proven it sound [29]. Parametric polymorphism and multiple dispatch naturally coexist, as long as dynamic dispatch is disallowed on an argument whose static type is a type parameter (but dispatch on an argument whose static type is a polymorphic class or interface, like `List<T>`, is safely supported). We believe this restriction naturally generalizes to safely handle predicate dispatch.

3.2 Modularly Typesafe Interface Dispatch

As with the MultiJava version in Figure 1, the JPred code in Figure 3 is only modularly typesafe if the AST nodes are classes. If instead the nodes are defined as in Figure 2, then JPred's modular type system must assume that there could exist a class implementing both the `If` and `While` interfaces, and therefore that the first and second methods in Figure 3 are potentially ambiguous. For this reason, our original version of JPred [28] simply disallowed interface dispatch altogether, as in MultiJava.

In this paper, we extend JPred to support interface dispatch while preserving modular typechecking. Our solution 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. For example, one way to resolve the ambiguity in Figure 3 is to add the following method:

```
Node optimize(Node n) when n@If && n@While { ... }
```

Because this new method overrides both the second and third methods and is applicable whenever both of them are applicable, the ambiguity between those two methods can never manifest itself as a run-time *message-ambiguous* error.

As an alternative to adding a dedicated method to handle the ambiguity, predicate dispatch also easily 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 `optimize` method to be `n@While`

```
class Optimizer {
  Node optimize(Node n)
    when n@If { ... }
  | when n@While { ... }
  | { return n; }
}
```

Figure 4. Ordered dispatch in JPred.

```
class Optimizer {
  Node optimize(Node n) when n@If { ... }
  Node optimize(Node n) when n@While && !n@If { ... }
  Node optimize(Node n) when !n@If && !n@While
    { return n; }
}
```

Figure 5. The desugared version of Figure 4.

`&& !n@If`, thereby explicitly indicating that nodes implementing both `If` and `While` should be optimized with the first `optimize` method; the second method is no longer applicable. With this revised predicate, the first and second methods are now disjoint and hence modularly guaranteed to be unambiguous.

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).

The ability for programmers to resolve ambiguities via predicate dispatch provides a straightforward approach to incorporating interface dispatch in JPred while preserving modular type safety. We have modified the JPred compiler to allow dispatch on interfaces. However, unlike the prior approaches to interface dispatch described in Section 2.3, we have not imposed any new restrictions or modifications to either the method-lookup semantics or to the type system. Instead we continue to use the original ambiguity-checking algorithm for JPred [28]. It is up to the programmer to resolve any signaled ambiguities in the manner deemed most appropriate. We have shown two common ways for programmers to resolve ambiguities, but variations on these approaches are possible.

3.3 Textual Order as Syntactic Sugar

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 [32], 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 4 illustrates our syntactic sugar, which we refer to as *ordered dispatch*, using a revised version of the `Optimizer`. 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 4 is modularly typesafe even if the AST nodes are interfaces. In particular, if the given node implements both `If` and `While`, the first case in the figure will be invoked.

However, unlike prior approaches that resolve ambiguities using textual order [6], the introduction of ordered dispatch does not entail any modifications to JPred's method-lookup semantics, since ordered dispatch is purely syntactic sugar. For example, Figure 5 shows the desugared version of the code in Figure 4. More generally, an ordered dispatch of the form

$$T\ m(\overline{T}\ \overline{x})\ \text{when}\ P_1\ \{\ \ldots\ \}\ |\ \cdots\ |\ \text{when}\ P_n\ \{\ \ldots\ \}$$

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

$$T\ m(\overline{T}\ \overline{x})\ \text{when}\ P_1\ \{\ \ldots\ \}$$
$$T\ m(\overline{T}\ \overline{x})\ \text{when}\ P_2\ \&\&\ !P_1\ \{\ \ldots\ \}$$
$$\ldots$$
$$T\ m(\overline{T}\ \overline{x})\ \text{when}\ P_n\ \&\&\ !P_1\ \&\&\ \cdots\ \&\&\ !P_{n-1}\ \{\ \ldots\ \}$$

Therefore, programmers can also easily mix ordered dispatch with regular JPred methods. As a simple example, in figure 4 the programmer could choose to make the last case in the ordered dispatch declaration a separate JPred method.

Because ordered dispatch is a syntactic sugar, the existing modular checks for exhaustiveness and unambiguity in JPred's typechecker are sufficient to ensure type safety in the presence of ordered dispatch. However, it is additionally useful to warn programmers when a case in an ordered dispatch declaration is unreachable, as this likely indicates a programming error. Such warnings are analogous to the *match redundant* warnings provided by Standard ML [32].

To support these kinds of warnings, we have augmented the JPred typechecker to require that each method's predicate be *satisfiable*. To see how this check subsumes a check for unreachable cases, consider the following ordered dispatch declaration:

```
void m(Object o) when o@Number { ... }
               | when o@Integer { ... }
```

Assuming that `Integer` is a subtype of `Number`, the second case above is unreachable. The JPred typechecker correctly signals a warning, because the desugared version of the second case's predicate is `o@Integer && !o@Number`, which is unsatisfiable. The satisfiability check is also useful for finding errors in regular JPred methods.

3.4 Implementation

The JPred compiler was implemented as an extension to the Polyglot [36] compiler for Java. Modifying the JPred compiler to support our approach to interface dispatch proved to be quite straightforward. Interface dispatch was already allowed by the parser; we added the ordered dispatch syntax, which the parser desugars as described earlier. We also modified the typechecks on individual predicates. First, we removed a typecheck that required dispatched-upon types to be classes. Second, we removed a check that each dispatched-upon type must be a strict subtype of the associated static type, which is too restrictive in the presence of multiple inheritance. Instead we now employ Java's *casting conversion* rules [24] to ensure the appropriate relationship: an expression of the static type must be able to be cast to the dispatched-upon type.

One novelty of the JPred compiler [28] is its usage of the CVC Lite [17] validity checker in order to precisely reason about predicates. For example, to decide whether a method with predicate P_1 overrides a method with predicate P_2, the JPred compiler asks CVC Lite whether the logical formula $P_1 \Rightarrow P_2$ is valid. We have augmented the JPred compiler to perform the satisfiability check for a predicate `P` similarly, by asking CVC Lite whether the formula `!P` is valid and signaling a warning if so. CVC Lite queries are also employed in the exhaustiveness and unambiguity typechecks on messages. These typechecks are described in detail in our earlier paper about JPred, and incorporating interface dispatch requires no changes whatsoever to their implementations.

As context for each query to CVC Lite, the JPred compiler generates a set of *axioms* that provide necessary semantic information. The original version of JPred encodes the subtype relation among classes by generating one conceptual axiom per pair of classes $\texttt{C}_1$ and $\texttt{C}_2$ appearing in a query [28]:

- If $\texttt{C}_1$ is a subclass of $\texttt{C}_2$, we declare the axiom $\forall \texttt{x}.(\texttt{x@C}_1 \Rightarrow \texttt{x@C}_2)$.
- Otherwise, if $\texttt{C}_2$ is a subclass of $\texttt{C}_1$, we declare the axiom $\forall \texttt{x}.(\texttt{x@C}_2 \Rightarrow \texttt{x@C}_1)$.
- Otherwise, we declare the axiom $\forall \texttt{x}.(\neg(\texttt{x@C}_1 \wedge \texttt{x@C}_2))$

The last kind of axiom above encodes the fact that classes support only single inheritance: if neither class is a subclass of the other, then there can be no object that is an instance of both.

In the presence of interface dispatch, we must incorporate information about interfaces into the axioms about the subtype relation. We therefore modify the above process to generate one axiom per pair of *types* appearing in a query, where each type is now either a class or an interface. The new process is identical to that described above, except that no axiom is generated in the last case if at least one of the types is an interface. Simply omitting that axiom forces CVC Lite to assume the possibility of multiple inheritance and therefore to treat interfaces conservatively.

We stress that our new approach to modularly typesafe interface dispatch is completely orthogonal to JPred's usage of CVC Lite to reason about predicates. The other existing languages that include predicate dispatch [19, 43, 37] employ their own specialized algorithms for reasoning about predicates. These algorithms could be modified to handle interface dispatch in much the same way as described above.

Finally, the JPred compiler's original code generation strategy is completely unchanged. The compiler generates ordinary Java source code. For each group of predicate methods that belong to the same message and are declared in the same class, a single Java method is generated that uses a linear sequence of `if` statements to determine which predicate method to invoke. The `if` statements are generated in some total order consistent with the method overriding partial order, from most-specific to least-specific. This algorithm is simple and modular, but it can cause unnecessary re-evaluation of portions of predicates from one `if` branch to the next. The focus of our work is on the problem of modular typechecking, which is orthogonal to code-generation issues. We could adopt work by others on generating efficient dispatch functions for predicate dispatch [13] without affecting our results on modular ambiguity checking for interface dispatch.

4. Featherweight JPred

This section overviews Featherweight JPred (FJPred), an extension of Featherweight Java (FJ) [25] that formalizes JPred's approach to interface dispatch. 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 [21].

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 [10, 9, 30, 29]. The original work on predicate dispatch [19] presented a formalization of predicate dispatch but did not prove type soundness.

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

Figure 6. The syntax of FJPred.

4.1 Syntax

Figure 6 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. We comment on these things 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 name 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:

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

Having all methods for a given message in one declaration simplifies the formal semantics. Note that the semantics of 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 interact in interesting ways with interface dispatch, 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 (see below).

4.2 Reasoning About Predicates

Both the dynamic and static semantics must reason about predicates and the relationships among them. The dynamic semantics must evaluate predicates and must know the overriding relation for methods, which depends on predicate implication, in order to perform method lookup. The static semantics must reason about predicates to check exhaustiveness and unambiguity.

Figure 7 provides the rules for evaluating predicates; the judgment $TT;\Gamma \models \texttt{P}$ formalizes the conditions under which a predicate evaluates to `true`. The rules use an explicit type table in the context, which shadows the implicit global type table; the reason for this will be clear below. As usual, Γ denotes a type environment, which maps variables to types. Intuitively, Γ provides the run-time classes of the actual arguments to a method, which is necessary to determine if a dynamic dispatch in the method's predicate succeeds. The rules rely on the subtype relation among types, denoted $TT \vdash \texttt{S<:T}$, which is straightforward.

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

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

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

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

Figure 7. Evaluating predicates.

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

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

Figure 8. Predicate validity.

To formalize reasoning about predicates, we essentially need to model the role that CVC Lite plays in the JPred compiler. It is beyond the scope of this formalization to formally model the particular decision procedures used by CVC Lite in order to prove a query valid. Instead, we formalize the *consequence* of a CVC Lite validity query. Figure 8 defines our notion of validity. The judgment $\overline{\mathtt{x}} \models \mathtt{P}$ indicates that a logical formula `P`, which uses the same syntax as FJPred predicates, is valid, where $\overline{\mathtt{x}}$ 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 $\neg(\mathtt{x@If} \wedge \mathtt{x@While})$, where `If` and `While` 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 to be valid. Quantifying over all extensions of TT formalizes the conservative nature of the validity check: CVC Lite must always assume the possibility of a class that implements both `If` and `While`.[1]

4.3 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 associated rules are straightforward adaptations of the FJ rules to our extended syntax. The only interesting rule is the one for method invocation:

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

The *mbody* function performs method lookup, given the run-time classes of the receiver and the other arguments. The rules defining *mbody* and a helper function are defined in Figure 9. These rules formalize the encapsulated style of multimethod dispatch [8],

[1] 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 CVC Lite is given information about. Our technical report [21] formalizes this approach, but the two notions of validity can be shown to be equivalent, so we employ the simpler version here.

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

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

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

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

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

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

Figure 9. Method lookup rules.

whereby all methods in a class are considered to override the inherited methods from superclasses. This is the semantics employed by the full JPred language and compiler. It would be a small change to instead formalize the symmetric semantics [11], and this change would not affect our results.

The first rule for *mbody* in Figure 9 applies when the receiver class contains a unique 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 given message but none are applicable, or there are no declared methods for the given message. In this case, lookup proceeds recursively in the direct superclass.

4.4 Static Semantics

As usual, the typechecking rules for expressions are formalized by a judgment of the form $\Gamma \vdash \mathtt{t} : \mathtt{T}$. The associated rules are straightforward adaptations of the FJ rules to our extended syntax. This includes the rule for typechecking message sends:

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

As in FJ, the *mtype* helper function looks up the type of a method in some class, searching the superclass if no method declaration is found. There is no need to search superinterfaces, because the exhaustiveness typecheck (described below) ensures that at least one implementation of the method exists. We augment the *mtype* function with the obvious rules for looking up the type of a method

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

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

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

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

$$\frac{\texttt{x} \in \overline{\texttt{x}}}{\overline{\texttt{x}} \vdash \texttt{x@S OK}} \qquad \frac{\overline{\texttt{x}} \vdash \texttt{P OK}}{\overline{\texttt{x}} \vdash \neg\texttt{P OK}}$$

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

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

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

Figure 10. Typechecking methods.

inside an interface, nondeterministically searching one of the superinterfaces if no method signature is found.

The top rule in Figure 10 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.[2] 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 by the bottom rule in the figure. The rule for ambiguity checking first requires that the two given predicates $\texttt{P}_1$ and $\texttt{P}_2$ are not logically equivalent unless they are the same textual predicate. The second premise uses a comprehension notation to collect the subset $\overline{\texttt{Q}}$ of predicates defined in the current method declaration that override both $\texttt{P}_1$ and $\texttt{P}_2$. The final premise then ensures that $\overline{\texttt{Q}}$ is a *resolving set* for $\texttt{P}_1$ and $\texttt{P}_2$: whenever both $\texttt{P}_1$ and $\texttt{P}_2$ are satisfied, then so is at least one predicate in $\overline{\texttt{Q}}$. Two special cases of this last requirement are worth noting. First, if $\texttt{P}_1$ overrides $\texttt{P}_2$, then $\texttt{P}_1$ is in $\overline{\texttt{Q}}$ so the last premise holds and the methods are considered unambiguous, and similarly for the case when $\texttt{P}_2$ overrides $\texttt{P}_1$. Second, if $\texttt{P}_1$ and $\texttt{P}_2$ are disjoint, then $\texttt{P}_1 \wedge \texttt{P}_2$ 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 11. 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) superclasses and superinterfaces. Each of the associated messages is then checked for proper method overriding and for exhaustiveness.

[2] For simplicity, our formalism does not model other typechecks on predicates, for example the casting conversion rules for dynamic dispatches discussed in Section 3.4.

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

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

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

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

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

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

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

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

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

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

Figure 11. Typechecking classes and interfaces.

The rules for method overriding are shown in the middle of Figure 11. Analogous with the rule for method overriding in FJ, we require that a class's superclasses and superinterfaces agree with the class on each method's type. We also require a class's superclasses and superinterfaces to agree with the class on each method's formal-parameter names, which are accessed by the *mformals* helper function. 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 11. The *mpreds* function returns all predicates associated with a method of the given message name in the given class and in all superclasses. The message is deemed exhaustive if the disjunction of all of these predicates is valid. This condition ensures that every invocation of the message will have at least one applicable method.

It is important for soundness that the *allMethodNames* function (definition not shown) returns all inherited method names in addition to the names of methods declared in the current class, so that inherited method names are subject to the overriding and exhaustiveness checks. For example, this ensures that if a class does not define a method m but inherits two different declarations of m, then these declarations will be required to agree on the method's type. As another example, considering inherited method names ensures that exhaustiveness checking will fail if a class inherits a method signature from some superinterface but does not provide or inherit an implementation of this method.

4.5 Type Soundness

We have proven a type soundness theorem for FJPred using the standard "progress and preservation" style [44].

THEOREM 4.1. (Progress) If $\vdash$ t : T, then either t is a value, t contains a subexpression of the form (S)(new C($\overline{\text{v}}$)) where $TT \vdash$ C$\not<:$S, or there exists some term s such that t $\longrightarrow$ s.

THEOREM 4.2. (Type Preservation) If $\Gamma \vdash$ t : T and t $\longrightarrow$ s, then there exists some type S such that $\Gamma \vdash$ s : S and $TT \vdash$ S<:T.

The full proofs of these theorems are available in our companion technical report [21]. 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.

5. Case Studies

This section describes two case studies we undertook to evaluate the effectiveness of JPred's interface dispatch in realistic settings. First, we updated the JPred compiler, which is written in JPred, to use interface dispatch instead of class dispatch. Second, we updated portions of Eclipse, which is written in Java, to use JPred.

5.1 JPred

The JPred compiler is built as a 15kloc extension to the Polyglot extensible Java compiler [36], which is written in Java. Polyglot uses a hierarchy of Java interfaces to represent the various AST nodes, and a parallel hierarchy of Java classes implements these interfaces. The intent is that extenders of the compiler never directly manipulate the underlying classes, instead only accessing them through the associated interfaces. This level of indirection is critical for ease of extension and for composition of extensions.

Our extension to Polyglot implementing JPred is itself written in JPred. There are several natural opportunities for employing predicate dispatch in the implementation of a Polyglot extension, most notably in the code for a new compiler pass. Polyglot supports the easy addition of new traversals (called *visitors* in Polyglot, by analogy with the visitor design pattern [22]) over the 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. In our new visitors, we used JPred to allow the dispatch constraints to be declaratively specified and statically checked for exhaustiveness and unambiguity, similar to the style illustrated by our hypothetical Optimizer in Figure 3.

Unfortunately, when we originally implemented our Polyglot extension, JPred did not support interface dispatch. Therefore, the only way to obtain the benefits of predicate dispatch was to dispatch directly on AST node classes instead of the associated interfaces. In this way, JPred's limitation forced us to violate the intended Polyglot style, subverting the provided abstraction layer. Dispatching directly on the underlying node classes also made our visitors extremely brittle in the face of later evolution or extension to the compiler.

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

Figure 12. A simple usage of class dispatch in the JPred compiler.

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

Figure 13. Interface dispatch version of Figure 12.

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

Figure 14. An example with multiple predicate methods.

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

Figure 15. Interface dispatch version of Figure 14.

After we added support for interface dispatch in the JPred compiler, we were able to rewrite the entire compiler to exclusively employ interface dispatch instead of class dispatch for the purposes of dispatching on AST nodes. In total, there were 28 messages whose method implementations were converted from using class dispatch to using interface dispatch. In 14 of these cases, the message 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 12 shows some code using class dispatch, and Figure 13 shows the version modified to employ interface dispatch.

The other 14 messages we modified each contained between two and 12 predicate methods, with a median of five. To handle these messages, we converted class dispatches to interface dispatches as shown above, and we additionally used the ordered dispatch syntactic sugar to allow modular ambiguity checking to succeed. Figure 14 shows a simple example involving two predicate methods, and Figure 15 shows the version modified to employ interface dispatch.

In Figure 15, 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 program error. If desired, the programmer can catch such errors at run time by adding a new method with predicate e@Unary && e@Binary that appropriately handles the erroneous scenario.

	Compile time (secs)	CVC Lite queries
JPred-orig	45.9	217
JPred-interface	47.3	310

Figure 16. Quantitative Results

However, that 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, as does an approach based on parasitic methods, which were discussed in Section 2.3. The approaches to interface dispatch embodied by Dubious and Half & Half, also discussed in that section, would alleviate this problem to some extent by imposing package-level restrictions that would make it easier for programmers to manually find all improper usages of multiple inheritance. However, these restrictions would also be impractical to abide by in the context of Polyglot. Dubious would disallow Polyglot extensions from ever multiply inheriting from existing AST node interfaces, since Polyglot extensions are written in their own package. Half & Half's restriction would cause the code in Figure 15 to fail to typecheck, since both `Unary` and `Binary` are declared `public`.

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 3.3 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, Figure 16 contains the quantitative results comparing the compilation of the previous 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 CVC Lite in order to ensure unambiguity, causing a small increase in overall compile time.

5.2 Eclipse

Eclipse is a widely used, extensible platform designed for building integrated development environments (IDEs), written in Java. Eclipse is structured as a small kernel, the *Platform Runtime*, and a collection of plugins that provide Eclipse's functionality, which are discovered at run time. We performed a small case study on a portion of Eclipse to evaluate JPred's utility for complex programs not designed with predicate dispatch in mind. For this study we updated the `getChildren()` and `hasChildren()` methods of all classes meeting the interface `ITreeContentProvider` in the Java Development Tooling (JDT) UI plugin, `org.eclipse.jdt.ui`. JDT adds Java support to Eclipse, and the `ITreeContentProvider` is used to display tree-structured information. Figure 17 shows the classes we modified.

Figures 18 and 19 show one of the simpler updates, which is representative of our general approach. Eclipse's plugin nature results in code that is heavily dependent on interfaces: both `IJavaModel` and `IProject` are interfaces. The original method performs manual interface dispatch via `instanceof` tests, along with associated run-time type casts. The JPred version

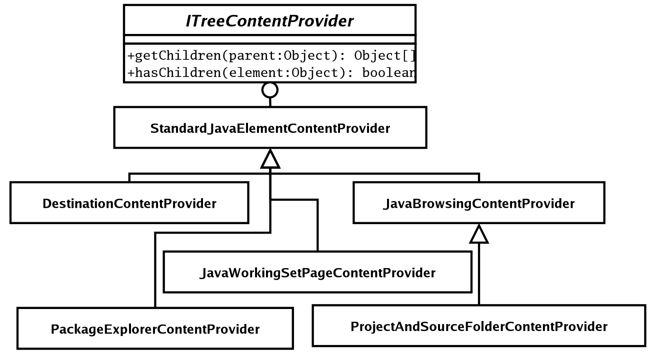

Figure 17. Eclipse's JDT UI `ITreeContentProvider` interface and its implementations.

```
public Object[] getChildren(Object parentElement) {
    try {
       if (parentElement instanceof IJavaModel)
         return concatenate(
             super.getChildren(parentElement),
             getNonJavaProjects((IJavaModel)parentElement));
       if (parentElement instanceof IProject)
         return ((IProject)parentElement).members();
       return super.getChildren(parentElement);
    } catch (CoreException e) {
       return NO_CHILDREN;
    }
}
```

Figure 18. An example method from Eclipse.

```
public Object[] getChildren(Object parentElement) {
    try {
       return getChildrenHelper(parentElement);
    } catch (CoreException e) {
       return NO_CHILDREN;
    }
}

protected Object[] getChildrenHelper(Object parentElement)
         throws CoreException
  when parentElement@IJavaModel {
    return concatenate(
         super.getChildren(parentElement),
         getNonJavaProjects(parentElement));
}
| when parentElement@IProject {
    return parentElement.members();
}
| {
    return super.getChildren(parentElement);
}
```

Figure 19. Interface dispatch version of Figure 18.

is more declarative, is checked for exhaustiveness and unambiguity, and obviates the need for type casts. Other updates relied on combinations of interface dispatch with other aspects of predicate dispatch. For example, the JPred version of `JavaBrowsingContentProvider.getChildren()` contains the following case, where `fProvideMembers` is a `boolean` field inherited from a superclass:

```
| when (fProvideMembers && element@IType) {
    return getChildren(element);
```

During the course of the case study, the JPred compiler revealed one error and one potential error in Eclipse's JDT UI plugin. First, `ProjectAndSourceFolderContentProvider.getChildren()` has two cases in an `if` statement that dispatch on whether the method's parameter implements the `IStructuredSelection` interface; the second case is unreachable. Updating this method to use JPred revealed the error because the (desugared version of the) second case's predicate was not satisfiable, causing a compile-time warning to be signaled. Second, `DestinationContentProvider.getChildren()` contains a sequence of `if-else if` statements that perform `instanceof` tests on interfaces, but without a final `else` clause. When this sequence was converted to a JPred helper method, a compile-time exhaustiveness error was signaled, forcing the programmer to be explicit about the intended behavior when no case applies. In the original code, it is unclear if such a situation is truly intended to be a no-op or if it is indicative of a program error.

This case study also illustrated some limitations of JPred. All of these limitations are orthogonal to our approach to interface dispatch. First, JPred only allows dispatch to occur at "top level." This limitation sometimes necessitated the creation of helper methods, like `getChildrenHelper` in Figure 19. It would be useful to explore a version of predicate dispatch that can be used within method bodies, analogous to a `switch` statement. Second, sometimes JPred's predicate language was too restrictive. For example, the `hasChildren` method in `JavaWorkingSetPageContentProvider` is implemented as a sequence of `if` statements whose first one is as follows:

```
if (element instanceof IProject &&
    !((IProject)element).isAccessible())
  return false;
```

It would be nice to convert this code to use predicate dispatch, but JPred does not allow method calls in predicates. We plan to augment JPred to support calls to methods that are declared (and checked to be) `pure`, meaning that they are side-effect-free. Finally, JPred's style is targeted to methods that can be implemented as several logically independent cases. There were some Eclipse methods whose structure was more complicated, for example depending on one case falling through to the next one. In these situations, it was not natural to employ predicate dispatch, so we left the code as is.

In total, the original Eclipse methods contained 30 type casts, all on interfaces. Updating these methods to use JPred reduced the number of type casts to three. One of the remaining casts relies on the relationship between a field's value and the type of a parameter. The other two remaining casts involve scenarios where JPred's limitations, discussed above, precluded predicate dispatch.

6. Conclusion

We have demonstrated a natural approach to resolving the tension between multiple dispatch and multiple inheritance while retaining fully modular static typechecking. The key idea is to move from multiple dispatch to the more general concept of predicate dispatch, whose extra expressiveness allows multiple-inheritance ambiguities to be modularly resolved by programmers. We have instantiated our approach to support dispatch on interfaces in JPred, a predicate-dispatch extension to Java that originally disallowed dispatch on interfaces altogether. We have validated our approach by formalizing JPred's interface dispatch and proving an associated type soundness theorem, and by demonstrating the utility of JPred's interface dispatch in two case studies.

References

[1] R. Agrawal, L. G. DeMichiel, and B. G. Lindsay. Static Type Checking of Multi-Methods. In *Proceedings of the OOPSLA '91 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 113–128, Nov. 1991.

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

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

[4] L. Bettini, S. Capecchi, and B. Venneri. Double Dispatch in C++. *Software – Practice and Experience*, 2005.

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

[6] J. Boyland and G. Castagna. Parasitic methods: Implementation of multi-methods for Java. In *Conference Proceedings of OOPSLA '97, Atlanta*, volume 32(10) of *ACM SIGPLAN Notices*, pages 66–76. ACM, Oct. 1997.

[7] K. Bruce, L. Cardelli, G. Castagna, The Hopkins Objects Group, G. T. Leavens, and B. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):217–238, 1995.

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

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

[10] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, Feb. 1995.

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

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

[13] C. Chambers and W. Chen. Efficient multiple and predicate dispatching. In L. Meissner, editor, *Proceeings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34(10) of *ACM Sigplan Notices*, pages 238–255, N. Y., Nov. 1–5 1999. ACM Press.

[14] C. Chambers and G. T. Leavens. BeCecil, a core object-oriented language with block structure and multimethods: Semantics and typing. In *4th Workshop on Foundations of Object-Oriented Languages*, Jan. 1997.

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

[16] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications*, volume 35(10) of *ACM SIGPLAN Notices*, pages 130–145, Oct. 2000.

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

[18] Eclipse home page. `http://www.eclipse.org`.

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

[20] N. Feinberg, S. E. Keene, R. O. Mathews, and P. T. Withington. *The Dylan Programming Book*. Addison-Wesley Longman, Reading, Mass., 1997.

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

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

[23] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA., 1983.

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

[25] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.

[26] G. T. Leavens and T. D. Millstein. Multiple dispatch as dispatch on tuples. In *OOPSLA '98 Conference Proceedings*, volume 33(10) of *ACM SIGPLAN Notices*, pages 374–387, Oct. 1998.

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

[28] T. Millstein. Practical predicate dispatch. In *OOPSLA 2004 Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Oct. 2004.

[29] T. Millstein, C. Bleckner, and C. Chambers. Modular typechecking for hierarchically extensible datatypes and functions. *ACM Transactions on Programming Languages and Systems*, 26(5):836–889, Sept. 2004.

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

[31] T. Millstein, M. Reay, and C. Chambers. Relaxed MultiJava: Balancing extensibility and modular typechecking. In *Proceedings of the 2003 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Anaheim, CA, Oct. 2003.

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

[33] D. A. Moon. Object-oriented programming with Flavors. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, pages 1–8. ACM Press, 1986.

[34] W. B. Mugridge, J. Hamer, and J. G. Hosking. Multi-methods in a statically-typed programming language. In P. America, editor, *ECOOP '91: European Conference on Object-Oriented Programming*, LNCS 512, pages 307–324. Springer-Verlag, 1991.

[35] Nice home page. `http://nice.sourceforge.net`.

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

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

[38] A. Paepcke. *Object-Oriented Programming: The CLOS Perspective*. MIT Press, 1993.

[39] A. Shalit. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language*. Addison-Wesley, Reading, Mass., 1997.

[40] J. Smith. Draft proposal for adding Multimethods to C++. Available at `http://std.dkuug.dk/jtc1/sc22/wg21/docs/papers/2003/n1529.html`.

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

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

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

[44] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 15 Nov. 1994.