Parasitic Methods: An Implementation of Multi-Methods for Java

John Boyland*
Computer Science Department,
Carnegie Mellon University,
5000 Forbes Avenue,
Pittsburgh, PA 15213-3891, USA
E-mail: `john.boyland@acm.org`

Giuseppe Castagna
CNRS, LIENS,
École Normale Supérieure,
45 rue d'Ulm,
75005 Paris, France.
E-mail: `castagna@dmi.ens.fr`

Abstract

In an object-oriented programming language, method selection is (usually) done at run-time using the class of the receiver. Some object-oriented languages (such as CLOS) have *multi-methods* which comprise several methods selected on the basis of the run-time classes of all the parameters, not just the receiver. Multi-methods permit intuitive and type-safe definition of binary methods such as structural equality, set inclusion and matrix multiplication, just to name a few. Java as currently defined does not support multi-methods. This paper defines a simple extension to Java that enables the writing of "encapsulated" multi-methods through the use of *parasitic* methods, methods that "attach" themselves to other methods. Encapsulated multi-methods avoid some of the modularity problems that arise with fully general multi-methods. Furthermore, this extension yields for free both covariant and contravariant specialization of methods (besides Java's current invariant specialization).

Programs using this extension can be translated automatically at the source level into programs that do not; they are modular, type-safe, and allow separate compilation.

1 Introduction

In an object-oriented language, method selection is (usually) done at run-time using the class of the receiver. Some object-oriented languages (such as CLOS [DG87, Ste90]) have *multi-methods* (also known as *generic functions*) which comprise several methods selected on the basis of the run-time classes of all the parameters, not just the receiver. Unfortunately Java, like its predecessor C++, does not provide multi-methods. Stroustrup, the designer of C++, regrets that he was unable to consider providing multi-methods in his language, essentially because, as he admits, he was not able to find how to do it (see Section 13.8 of [Str94]). With this work, we demonstrate that this situation can be avoided for Java.

In this paper, we show how one can implement the special case of "encapsulated" multi-methods for Java. While specific to Java, our ideas can be applied with little change to other languages. Our solution does not implement multi-methods in their most general form, since we want to preserve two of the most important properties of Java: modularity and separate compilation. Another goal is that our extension should be *conservative*, that is, have no effect on existing Java programs. To these ends, we adapt the well-established technique of *encapsulated* multi-methods (one of the solutions proposed by Bruce *et al.* [BCC+96] for typing binary methods). We call the enabling technical feature *parasitic* methods.

A *parasitic* method is a Java method in its own right which, *additionally*, extends the functionality of other methods (the *host* methods) for certain argument cases. If a host method is called with arguments that fit the parasitic method's parameter types, the parasitic method body is called instead of the host method body. A parasitic method usually *covariantly specializes* host methods, that is, handles some subset of the cases handled by the hosts. In earlier work [BC96], we have shown that covariant specialization is type safe, if, as with the parasitic methods described here, the new method only *partially overrides* the original method, that is, it overrides the method only for those arguments it can handle. A parasitic method may also *contravariantly specialize* a host method, that is, handle any possible set of arguments passed to the host, and handle other cases as well. In this case, the host method body will never be executed. Analogous to a mistletoe plant that draws nourishment from its host oak tree, a parasitic method draws from its host methods calls that the parasite can handle (according to the dynamic types of the arguments). Parasitic methods reconcile covariant and contravariant specialization in a single simple framework, while standard Java allows only invariant specialization (that is, a method can be overridden only by a method defined for the same parameter types), without sacrificing type safety or separate compilation.

Extending Java with parasitic methods is conservative; the extension does not affect the typing or the semantics (or even the performance) of Java programs that do not contain parasitic methods. Indeed, we show later that parasitic methods can be considered as a sort of syntactic sugar since they can be directly translated into standard Java.

*Effort sponsored by the Defense Advanced Research Projects Agency, and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0241. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory or the U.S. Government.

The paper is organized as follows. In Section 2, we introduce parasitic methods using several examples. We then describe how they are inherited and overridden, and informally describe type-checking and run-time selection. The latter two topics are detailed precisely in Section 3. In Section 4, we demonstrate a translation of parasitic methods into standard Java. Some of the design issues we engaged are exposed in Section 5. We discuss practical implications in Section 6, review related work in Section 7, and present issues for further work in Section 8. A conclusion ends the body of the paper.

2 Parasitic Methods

In this section we describe parasitic methods informally. First we give several examples. Then we describe what hosts a parasite "attaches" to, what additional static checking rules parasitic methods entail, and which parasite is selected dynamically when a host with multiple parasites is called. We end the section with a discussion of how our proposal preserves modularity.

2.1 Examples

In this section, we show several different ways to use parasitic methods. We start by showing what we consider their "standard" use, namely the covariant specialization of another method. Next, we show how parasites can be used to express contravariant method overriding. Lastly, we show some static parasites.

Despite their different uses, all parasitic methods obey the same selection discipline: at compile-time the standard Java overloading resolution[1] is performed for every method invocation (ignoring whether methods are declared as parasites or not); at run time if a method is selected and has a parasite, it passes the control to it when the parasite is applicable to the run-time types of the method's arguments.

2.1.1 Covariant parasites

A method is declared *parasitic* using the method modifier `parasitic`. For a simple example of parasitic methods, consider a class `IntList` with a method `union` that takes as argument another instance of `IntList` and produces a list that contains all the elements of `this` (Java's identifier for message receiver), as well as all elements from the argument that do not already occur in `this`. The method might be implemented using a naive $O(mn)$ algorithm (where m and n are the lengths of the two lists), or perhaps by a more sophisticated $O(m \lg m + n \lg n)$ algorithm that sorted them first.

Later, we define a subclass `IntSortedList`, in which methods are overridden so that the instances of this class are always ordered. The method `union` can be inherited from `IntList`. However it is clear that computing the union of two ordered lists can be done more efficiently using a simple $O(m+n)$ merge. The merge is implemented in a parasitic method in the definition of `IntSortedList` as shown in Figure 1. In Figure 1, the class `IntSortedList` inherits a method `union` that works on general `IntList`'s and also defines a new more specific method that handles the case of unioning two sorted lists. Java uses compile-time overloading to select methods, and so if the static type of two lists is `IntSortedList`, the more efficient method will be selected. However, if one of the two lists being unioned has static type of `IntList`, the Java compiler will select the general, less efficient method. By making the more efficient method a *parasitic* method, the general method yields to the efficient method *at run-time*. Indeed when a method is declared parasitic it automatically parasitizes every method *less specific* than it (a method is less specific than a second method if it has the same name and number of parameters, and the type of each parameter is a supertype of the type of the corresponding parameter in the second method; informally, a method is less specific than a parasite if it handles every argument the parasite could handle). In the example then the method declared in `IntList` parasitizes the method inherited from `IntSortedList`. Technically, the parasite does not *override* the host (the inherited method). It "attaches" itself to the host and diverts calls away that the parasite can handle. In essence, before executing its body, the general method checks the run-time type of the argument and calls the specific method if it is applicable. (As with static method selection, `null` is assumed to be an instance of every class.)

```
class IntList {
  public IntList union(IntList l)
           {  body for unioning two lists   }
}

class IntSortedList extends IntList {
  public parasitic IntSortedList union(IntSortedList l)
           {  body for merging two ordered lists   }
}
```

Figure 1: Parasite of an inherited host method

In Figure 1, the method in the subclass parasitizes a method inherited from the superclass. Of course, it is also possible to parasitize a method that is defined in the same class. For instance, in our example, even if only one of the two lists to union is ordered, it is possible to execute more efficiently (if m is the length of the ordered list, we have an $O(n \lg n + m)$ algorithm). Thus an even more efficient solution would be to override the method inherited from `IntList` as well, as shown in Figure 2. In this case, when the argument of the method is an instance of `IntList`, the overriding version of `union` is executed.[2]

[1] The static class of the receiver and its superclasses are examined for accessible definitions of the method. The most specific method applicable to the static types of the arguments, if one exists, is chosen at compile time. Otherwise, the call is flagged as either being ambiguous or having no applicable method. In standard Java, the actual method to be invoked will be determined at run time, using dynamic method lookup (i.e., using the dynamic type of the receiver to select among the methods that override the statically selected method).

[2] In the example in Figure 2, the return type of overriding method `union(IntList l)` in class `IntSortedList` is a subclass of the return type of the method it overrides. This discipline of overriding was implemented in an early version of the Java compiler and Drossopoulou and Eisenbach have shown it to be type safe [DE97]. Thus, although the actual Java specification requires the result type of an overriding method to be the same as the result type of the method it overrides, we have chosen to adopt the less strict discipline.

```
class Union {
  public static parasitic IntList union(IntList l1, IntList l2)
      { return union(IntSortedList.sort(l1),IntSortedList.sort(l2)); }
  public static parasitic IntList union(IntList l1, IntSortedList l2)
      { return union(IntSortedList.sort(l1),l2); }
  public static parasitic IntList union(IntSortedList l1, IntList l2)
      { return union(l1,IntSortedList.sort(l2)); }
  public static parasitic IntSortedList union(IntSortedList l1, IntSortedList l2)
      {  body for merging two ordered lists  }
}
```

Figure 4: Static parasitic method example

```
class IntList {
  public IntList  union(IntList l)
             { body for unioning two lists }
}

class IntSortedList extends IntList {
  public IntSortedList  union(IntList l)
             { body for efficiently unioning two
               lists when the first is ordered }

  public parasitic IntSortedList  union(IntSortedList l)
             { body for merging two ordered lists }
}
```

Figure 2: Parasite of a local host method

```
interface IntBag {
       :
       :        // various method signatures
}

class IntList implements IntBag {
  public IntList union(IntList l) { ... }
       :
       :        // implementation of interface's signatures
}

class IntSortedList extends IntList {
  public parasitic IntSortedList union(IntBag b)
        { super general version }

  public parasitic IntSortedList union(IntSortedList l)
        { efficient specific version }
}
```

Figure 3: Contravariant Parasite

2.1.2 Contravariant parasites

In the examples above, all the parasites covariantly specialize their host, that is, they handle fewer arguments than their host. Our proposal gives the programmer the power of *contravariant specialization*, in which a new method handles *more* cases than the method it takes the place of. A parasite not only attaches to all less specific methods defined in the class (that is, declared or inherited), but also to any more specific inherited method. In the latter case, the host yields *all* control to the parasite, as if it had been overridden. For example, suppose in `IntSortedList`, we decided to implement `union` for any `IntBag` object, where `IntBag` is declared as an interface and that `IntList` implements it (see Figure 3). By declaring the general version of the method parasitic, we completely shadow the union method in `IntList`. When an instance of `IntSortedList` receives a union message, it executes the second method declared in its class if the argument is an `IntSortedList`, the first one otherwise.

2.1.3 Parasites for static methods

It is also possible to attach parasites to static methods, as long as the parasites are also static. In this case, all the methods must be declared in a single class, since static methods are not overridden in subclasses, only shadowed. Figure 4 shows how one could write a multi-method of four methods for unioning lists. Here, for uniformity, all methods are declared `parasitic`, although the first has no hosts. The static method `IntSortedList.sort` is used to sort a list and return an `IntSortedList` instance. The most specific method will be chosen at run-time.

2.2 Inheritance

When one class extends another class with parasitic methods, the parasite-host relation is inherited.

As with regular methods, a parasitic method is *overridden* whenever a method with the same name and the same parameter types is defined in the subclass. Otherwise, assuming the method is accessible, it is *inherited*.

When a parasitic method is overridden by a non-parasitic method, the new method remains implicitly parasitic, with exactly the hosts (and the parasites) it had before, except that if any of these methods is overridden by a method declared parasitic, the connection is broken.

For the sake of an example, we define in Figure 5 the class `IntSortedList`, a subclass of `IntList` from Figure 1, with three `union` methods for arguments of type `IntList`,

```
class IntSortedList extends IntList {
  public IntSortedList  union(IntList l)
             { body for efficiently unioning two
                lists when the first is ordered }

  public parasitic IntSortedList  union(IntSortedList l)
             { body for merging two ordered lists }

  public parasitic IntSortedList  union(IntSet l)
             { specialized body for IntSet }
}


class IntSet extends IntSortedList {
  public IntSortedList  union(IntSortedList l)
             { some specialized body }
}
```

Figure 5: Overridden parasite

IntSortedList, and IntSet respectively, together with a new subclass IntSet where the "middle" parasite of signature IntSortedList union(IntSortedList) is overridden. The parasite for IntSortedList has been overridden with a normal method, and thus it keeps the host it had before, that is the method union(IntList l). Furthermore the inherited method union(IntSet l) becomes a parasite of the method declared in IntSet. For example the command,

```
  (new IntSet()).union((IntList)new IntSortedList())
```

executes the "some specialized body" while

```
  (new IntSet()).union((IntSortedList)new IntSet())
```

executes the "specialized body for IntSet."

2.3 Intuitive hosts

As shown by the preceding examples, a method declared parasitic attaches itself to every less specific method declared in the class or inherited. It also parasitizes every *inherited* method that is more specific. Furthermore, a method not declared parasitic is nonetheless parasitic if it overrides a parasitic method. It has the hosts that this method had, with the exception of any methods that have subsequently been overridden by methods declared parasitic. In other words, the parasite-host relation is only changed when one of the two methods at issue is overridden by a method declared parasitic. This rule permits the body of the parasite to be changed without changing the parasite-host relation.

The inheritance of the parasite-host relation was one of the most difficult design issues of our work (see Section 5 later on). The actual machinery may seem overly complicated but, actually, it obeys a few simple principles. The reader/programmer must consider that all the parasitic methods with the same name and number of arguments together form a multi-method. These methods interact so that whenever one of them is called, the most specific (more precisely, a maximally specific) among those that can handle the arguments is executed. Now if we have a class with a multi-method and we want to define a subclass of it, we may wish either to inherit the multi-method as it is or to modify it. In the latter case, three possible modifications can be envisaged: (1) To replace (override) one or more methods of the multi-method; (2) to replace the whole multi-method by a brand new (multi-)method; (3) to add new methods to the multi-method.

The first case is obtained by the standard Java overriding; so if we want to override a particular method of the multi-method it suffices to declare in the subclass a new method with the same signature (with possibly smaller return type) but not declared parasitic.

The second case, defining a new multi-method, is obtained when a subclass declares one or more parasitic methods that are either more or less specific than some method of the old multi-method. The newly defined parasitic methods form a new multi-method that, in a sense, "partially overrides" the old multi-method. Roughly speaking, the methods of the old multi-method are called when no method of the new multi-method applies. In other words, a method is first looked for locally among the methods declared in the class and if none applies, the search continues in the old multi-method.

The addition of new methods (the third case) is straightforward when the methods to add are more specific than all the methods of the multi-method (which should be the most frequent case). As we have seen in many examples in this section, it suffices to declare the new methods parasitic. However, our system is less flexible when the method to add is less specific than some method of the multi-method, since this addition can be obtained only by overriding all the methods more specific than the new method with methods that perform super calls.

In conclusion, the programmer needs to remember that plain overriding causes the replacement of the method in the multi-method, while parasitic overriding (or declaration) entails a new multi-method partially overriding the old multi-method, if it exists.

2.4 Static type-checking and other rules

We add a single type rule to handle parasitic methods:

> *The result type of a parasite must be assignment compatible with the result type of its hosts.*

A parasite is called from the host if the run-time types of the arguments fit the parasite. The condition says it is a compile-time error if the return value of the parasite could not be returned by the host.

Further restrictions apply. The parasite cannot name an exception in its throws clause not mentioned in the host's throws clause. The parasitic method must be static if and only if the host method is static. An inherited host cannot be final (because we change its meaning in the subclass). For simplicity, we require the parasite to have the same accessibility as the host. For ease of implementation, we disallow parasites from being abstract, and hosts from being native.

2.5 Intuitive selection

When multiple parasites are attached to the same host, there is the issue of *priority*: which parasite is checked for applicability first. The priority closest to the semantics of most multi-method systems would be to use the most-specific parasite. However, requiring that such a parasite always exist poses many problems (see Section 2.6 for a detailed explanation). Thus we decided to use textual order as a tie-breaker: the textually last parasite that applies is chosen.

However, since parasites themselves can have other parasites, the most specific parasite (if it exists) will always be called, no matter what priority is used. If a less specific parasite is selected, then this parasite will be a host to more specific parasites which then have another "chance." Thus the semantics of parasites can be informally stated as follows:

> *When a host method with attached parasites is selected, the system searches for a parasite that is applicable to the run-time types of the method arguments.*
> *The search starts from the class of the method receiver and follows an ascending order. Namely, the parasites are checked within a class from the last defined to the first defined and every class is searched before its superclasses.*
> *The first parasite found to be applicable is executed in the place of the method. If no parasite applies, then the body of the method is executed.*

Thus textual order is a simple solution to the problem of how to choose the "most-specific" method, which must be addressed by any multi-method system.[3]

2.6 Modularity

As we said in the introduction we do not implement multi-methods in their most general form since we want to preserve two of the most important properties of Java: modularity and separate (type-checking and) compilation.

Textual order is one important ingredient. For example, if `I1` and `I2` are two unrelated interfaces, there is no harm in defining

```
class A {
  void m(Object x) { ... }
  parasitic void m(I1 x)  { ... }
  parasitic void m(I2 x)  { ... }
}
```

Indeed, if later a new class `B` that implements both `I1` and `I2` is added to the system, it is not necessary to modify the definition of `A`, since the conflict caused by an expression such as `(new A()).m((Object)new B())` is handled by the textual order.[4]

Consider what would happen if we had adopted the semantics that the most specific parasite were chosen, instead of using textual ordering. If there were such a class `B` in the final program that implemented both interfaces, then no most specific parasite would exist. Thus a "most specific parasite" rule would break modularity. Extending the type system with an explicit null type and new types of the form $T_1\&T_2$, representing the greatest lower bound of the two types, would permit a set of parasites to be checked separately. For instance, by adding a parasite

```
parasitic void m(I1&I2 x)
```

to the example, one could ensure the existence of a most specific parasite in all situations. But this solution would not only entail a significant change to the type system, but also require the writing of many parasites that may never be called.

The other ingredient for modularity is maintaining Java's restriction that all methods must be declared in the class of the receiver (`this` in Java). The price to pay is less flexibility in the use of multi-methods; parasitic methods achieve the full generality of multi-methods only if the writer of a class anticipates what subclasses will be derived from it, as explained by Bruce *et al.* [BCC+96]. For example in order to define all the possible cases of `union` in Figure 2, the class `IntList` should be defined as

```
class IntList {
   public IntList union(IntList l)
        { body for unioning two lists }
   public parasitic IntSortedList union(IntSortedList l)
        { body for efficiently unioning two lists
           when the second is ordered }
}
```

But then either the programmer writing the class `IntList` must already know that the class `IntSortedList` will be defined, or else the second method must be added to the class `IntList`, which then must be recompiled. In more general systems such as CLOS and Cecil, there is no such restriction on where multi-methods may be declared.[5]

It would be possible to define an extension of Java with general multi-methods, on the lines of the work [Cas97], but at the expenses of modularity. In languages that permit methods to be added to existing classes, parasitic methods would be equivalent to general multi-methods. Such a feature is compatible with separate compilation (as in O_2 [BDK92] where method addition and separate compilation coexist), but breaks modular type-checking.

3 Type-Checking and Selection

In this section, we define hosts, type rules, and parasite selection precisely. We begin with some definitions and then proceed to consider hosts, types and selection individually.

3.1 Formal definitions

We define two relations on types: *subtyping*, and a broader relation, *assignment subtyping*. A type T is a subtype of type T' (written $T \leq T'$) if $T = T'$ or if T is a descendant of class or interface T' (in case of array types, recursively apply this definition to the element types). We say T is a *strict subtype* of T' (written $T < T'$) if only the second condition holds. Additionally the *null type* is a strict subtype of all *reference types*.[6] A type T is an *assignment subtype* of type T' (written $T \leq_A T'$) if T is a subtype of T' or there is a primitive widening conversion (§5.1.2 in [GJS96]) from T to T'. For example, `int` is an assignment subtype of `float`, but not a subtype. Method invocation conversion (§5.3) is legal precisely from a type to an assignment supertype.

[3] Textual order thus plays somewhat the same role as class precedence lists in CLOS.

[4] Without the cast, the expression would be an *ambiguous method invocation* and would be rejected by the Java compiler.

[5] A similar observation can be made for general multi-method management. In languages such as CLOS and Cecil, new method definitions can be added to existing multi-methods at any time and anywhere in the source code. This can be simulated in our extension by adding new parasites in some particular classes, with a certain amount of recompilation.

[6] *Reference types* (§4.3 of [GJS96]) are class types, interface types, and array types. Together with the null type and *primitive types* (`boolean, byte, short, int, long, char, float,` and `double`) they comprise the types of Java.

Subtyping is extended to method signatures. Consider the following method definition:

```
    T  m(S1  x1, ... ,  Sn  xn)  {  ...   }
```

This method has *signature* $(S_1, \ldots, S_n) \rightarrow T$ or, in short, $\vec{S} \rightarrow T$. A type vector $\vec{S}$ is a subtype of $\vec{S'}$ if they have the same number of components and respective components are subtypes ($S_i \leq S'_i$). If at least one component is a strict subtype as well, the vector $\vec{S}$ is a strict subtype of $\vec{S'}$ (written $\vec{S} < \vec{S'}$).

A method signature $\vec{S} \rightarrow T$ is *more specific* than another method signature $\vec{S'} \rightarrow T'$ if and only if $\vec{S} < \vec{S'}$ (the return types are ignored).[7] Analogously, a signature $\vec{S} \rightarrow T$ is *less specific* than another method signature $\vec{S'} \rightarrow T'$ if and only if $\vec{S} > \vec{S'}$.

We say a method named m with signature $\vec{S} \rightarrow T$ is *defined* in a class C if it is declared in C or is inherited from a superclass or a superinterface. The existing Java type system ensures that for two different methods with the same name defined in a class C with signatures $\vec{S} \rightarrow T$ and $\vec{S'} \rightarrow T'$, we have $\vec{S} \neq \vec{S'}$.

3.2 Hosts

A *parasitic* method is one declared using the `parasitic` method modifier, or one overriding a parasitic method. A parasitic method m with signature $\vec{S} \rightarrow T$ defined in a class may have *hosts*, which are drawn from the set of methods defined in the class with the same name and number of arguments. We write

$$m(\vec{S}) \rightarrow T \hookleftarrow_C m(\vec{S'}) \rightarrow T'$$

if this parasite has a host with signature $\vec{S'} \rightarrow T'$ in the class C.

Definition 1 *Let C be a class, possibly extending a class B. $m(\vec{S}) \rightarrow T \hookleftarrow_C m(\vec{S'}) \rightarrow T'$ if and only if both methods are defined in C and one of the following conditions is satisfied:*

1. *A method m with signature $\vec{S} \rightarrow T$ is declared parasitic in class C and $\vec{S} < \vec{S'}$.*
2. *A method m with signature $\vec{S} \rightarrow T$ is declared parasitic in class C and a method m with signature $\vec{S'} \rightarrow T'$ is inherited from B and $\vec{S} > \vec{S'}$.*
3. *No method m with signature either $\vec{S} \rightarrow T$ or $\vec{S'} \rightarrow T'$ is declared parasitic in class C, and $m(\vec{S}) \rightarrow T \hookleftarrow_B m(\vec{S'}) \rightarrow T'$.*

An important property of this definition is that there can never be a cycle in the parasite-host relation within any class. New edges are only added between method signatures when one is declared parasitic, and the definition does not inherit *any* edge impacting such method signatures.

According to the definition above, a parasite is either more specific or less specific than its host. This means that the signatures of the two methods differ over some parameters with reference types but must be the same over parameters with primitive types (see Footnote 7). This restriction can be harmlessly weakened to permit parasites with primitive type arguments to have hosts where the corresponding arguments' types are assignment subtypes, but we preferred not to do so since it might have confused the programmer without bringing any significant enhancement.

It is possible to *declare* a method with a contravariant parasite:

```
class A {
  public void m(B x) { ... }
}

class B extends A {
  public parasitic void m(A x) { ... }
}

class C extends B {
  public void m(B x) { ... }
}
```

The method in `C` will never be executed since it is shadowed by the (inherited parasitic) method declared in `B`. The compiler issues warnings for such cases.

3.3 Typing

From the typing point of view, a method with signature $\vec{S} \rightarrow T$ having a host with signature $\vec{S'} \rightarrow T'$ must return an assignment subtype ($T \leq_A T'$). If the condition is not satisfied, the compiler generates an error message. If the involved parasite is inherited, then the compiler signals that the host's return type must be an assignment supertype of any parasite's return type. If the involved parasite is not inherited, then the compiler signals that the parasite's return type must be assignment compatible with the return type of its host.

3.4 Selection

Let us define more formally the discipline of selection of parasitic methods. For a class C let Σ^C_m be the set of signatures of the (possibly inherited) parasites of a method $m(\vec{S}) \rightarrow T$ in the class C:

$$\Sigma^C_m = \{\vec{S'} \rightarrow T' \mid m(\vec{S'}) \rightarrow T' \hookleftarrow_C m(\vec{S}) \rightarrow T\}$$

On Σ^C_m we define a total order $\prec^C_m$ as follows.

Definition 2 *Let C be a class, possibly extending a class B and consider $\vec{S_i} \rightarrow T_i, \vec{S_j} \rightarrow T_j \in \Sigma^C_m$. $\vec{S_i} \rightarrow T_i \prec^C_m \vec{S_j} \rightarrow T_j$, if and only if one of these conditions is satisfied.*

1. *the method for m with signature $\vec{S_i} \rightarrow T_i$ appears in class C after the definition of the method with signature $\vec{S_j} \rightarrow T_j$;*
2. *the methods for m with signature $\vec{S_i} \rightarrow T_i$ is declared in C while the method with signature $\vec{S_j} \rightarrow T_j$ is inherited from B;*
3. *the method for m with signature $\vec{S_i} \rightarrow T_i$ and the one with signature $\vec{S_j} \rightarrow T_j$ are both inherited from B, and $\vec{S_i} \rightarrow T_i \prec^B_m \vec{S_j} \rightarrow T_j$*

[7] The Java language specification uses the broader relation $\vec{S} <_A \vec{S'}$ for compile-time overloading resolution. Note that our choice implies that whenever one signature is more specific than another then the two signatures may differ for parameters with reference types but are the same for the parameters with primitive types.

The order relation $\prec_m^C$ on Σ_m^C is a total order, since it is a subrelation of the total order obtained by listing the ancestors of C in the (single) inheritance order and listing the methods within each class in order of declaration.

Then the selection discipline for parasites has the following definition:

Definition 3 (Selection) *When m in class C is applied to a tuple of arguments of type $\vec{S}$ then the method whose signature is* min *w.r.t.* $\prec_m^C$ *of* $\{\vec{S_i} \to T_i \in \Sigma_m^C \mid \vec{S} \leq \vec{S_i}\}$*, is then applied to the arguments. If this set is empty, the body of the host method is executed.*

This definition entails an iteration if the selected parasite itself has parasites. Since the definition of hosts ensures that the parasite-host relation never has cycles, this iteration must terminate.

4 Translation

Parasitic methods are easily translated to standard Java using that language's type testing primitives (see Figures 6 and 7), by adding conditional calls to the parasites at the beginning of the body of each non-abstract host method. First, the body of each parasite is packaged in a `final` (non-overrideable) method so that it can be called by the hosts. The name of this new method is formed by concatenating the class name, a dollarsign and the original method name. Next, if the host method is inherited (as in Figure 1), it is rewritten as an overriding method that simply calls `super`.[8] Then, a host's body is preceded by conditions that check applicability for each parasite in the inverse order they are declared in the class (tests for inherited parasites come last, as seen in Figure 8). As with standard Java, the null type is considered a subtype of every reference type. Only if none of the parasites is applicable, does control fall through to the original method body.

Some tests for applicability can be omitted. More precisely, in order to test that a parasitic method is applicable, the system need not consider a host's arguments when their types are assignment subtypes of the type of the corresponding parasitic method parameter. This situation occurs for contravariant parasites, as seen in Figure 9.

The generated argument checks are always legal since they are only needed to go down the inheritance hierarchy. The return statements type-check as long as the type condition described in Section 3.3 is satisfied.

Our modified compiler generates legal Java class files that can be verified and executed by a standard Java runtime system. It is even possible to use a standard Java compiler to compile clients or subclasses of classes using parasitic methods, but in this case of course, the parasite-host relation is not preserved in subclasses.

[8]Because of the definition of *more specific* used for static overloading resolution in standard Java, this step may introduce (or eliminate) ambiguous method resolution in the resulting program. Our implementation avoids this problem by translating directly to Java byte code. Even when compiling a client class using a class with parasitic methods, our implementation hides the generated methods from static dispatching, but if a standard Java compiler is used to compile the client, they become visible.

```
class IntList { ... } // unchanged

class IntSortedList extends IntList {
  public IntList union(IntList l) {
    if (l == null ||
        l instanceof IntSortedList) {
      return IntSortedList$union((IntSortedList)l);
    }
    else return super.union(l);
  }

  public IntSortedList union(IntSortedList l) {
    return IntSortedList$union(l);
  }

  public final IntSortedList
      IntSortedList$union(IntSortedList l)
    { body for merging two ordered lists }
}
```

Figure 6: Translation of Figure 1 (the host is inherited).

```
class IntList { ... } // unchanged

class IntSortedList extends IntList {
  public IntSortedList union(IntList l) {
    if (l == null ||
        l instanceof IntSortedList) {
      return IntSortedList$union((IntSortedList)l);
    }
    else { body for efficiently unioning two
           lists when the first is ordered }
  }

  public IntSortedList union(IntSortedList l) {
    return IntSortedList$union(l);
  }

  public final IntSortedList
      IntSortedList$union(IntSortedList l)
  { body for merging two ordered lists }
}
```

Figure 7: Translation of Figure 2 (the host is local).

```
class IntSet extends IntSortedList {
  public IntSortedList union(IntList l) {
    if (l == null || l instanceof IntSortedList) {
      return IntSet$union((IntSortedList)l);
    } else if (l == null || l instanceof IntSet) {
      // this branch will never be taken but it is checked
      // last since it corresponds to an inherited parasite
      return IntSortedList$union((IntSet)l);
    } else {
      return super.union(l);
    }
  }

  public IntSortedList union(IntSortedList l) {
    return IntSet$union(l);
  }

  public final IntSortedList
      IntSet$union(IntSortedList l) {
    if (l == null || l instanceof IntSet) {
      return IntSortedList$union((IntSet)l);
    } else { some specialized body }
  }
}
```

Figure 8: Translation of `IntSet` from Figure 5 (the parasite is overridden)

```
interface IntBag { ... } // unchanged

class IntList implements IntBag { ... } // unchanged

class IntSortedList extends IntList {
  public IntList union(IntList l) {
    if (l == null ||
        l instanceof IntSortedList) {
      return IntSortedList$union((IntSortedList)l);
    } else {
      return IntSortedList$union((IntBag)l);
    }
  }

  public IntSortedList union(IntBag b)
  { return IntSortedList$union(b); }
  public final IntSortedList
      IntSortedList$union(IntBag b)
  { super general version }

  public IntSortedList union(IntSortedList l)
  { return IntSortedList$union(l); }
  public final IntSortedList
      IntSortedList$union(IntSortedList l)
  { efficient specific version }
}
```

Figure 9: Translation of Figure 3 (contravariant parasite).

5 Design issues

The Java extension defined in this work is the result of a series of decisions. We summarize here the main design options, showing for each the pros (⊕) and, above all, the cons (⊖) that made us to reject them.

Multi-methods

1. Multi-methods could be added to Java by extending the language with CLOS's generic functions.

⊕ This would be the most powerful extension of Java with multi-methods and it subsumes all the others.

⊖ We excluded it from the very beginning since it would have broken modularity and separate compilation. In any case their power can be obtained from parasites by breaking modularity. So we preferred to make their use an exception rather than the standard.

2. We could have considered all methods as parasitic methods by default. This is what it is done in KOOL [Cas97] and what we proposed [BC96] for the database programming language O_2 (the goal being to make O_2's covariant specialization statically safe).

⊕ This solution is very clean and easy to understand. Practically speaking this solution consists of having dynamic (rather than static) resolution for Java overloaded methods.

⊖ It changes the semantics of existing programs that use overloading (but in our opinion it would be the best solution for a brand new language).

Parasites

3. Replace the use of the textual priority for the choice of parasites by a condition ensuring the existence of a best matching parasites. In other words, instead of performing parasite selection with an upward search for the the first parasite that can handle the arguments, select the parasite whose parameters best match the actual arguments.

⊕ This option was attentively considered since it is very intuitive and clean, and the conditions to ensure static type-safety are well-known (see [ADL91] or [Cas97]). Furthermore, by a clever compilation the average selection time can be logarithmic in the number of parasites (while with textual ordering it is linear).

⊖ A plain implementation of this discipline would break modularity as we have shown in Section 2.6. We considered a way to recover modularity in this case that consists of requiring the set of parasite signatures to always contain a least type. Such a requirement could not be met in many situations without also extending the type systems with an explicit null type and new types of the form $T_1\&T_2$, in which one of T_1 and T_2 is an interface type, representing the greatest lower bound of the two types. A parasite defined for the type $T_1\&T_2$ would then be executed when there is a selection conflict between T_1 and T_2. But besides the fact that we disliked the introduction of new types, this would have sometimes required the programmer to define a large number of parasites that would be never selected (typically, parasites defined for $T_1\&T_2$ where the T_i's are unrelated interfaces that will never have a common subtype) and could be very cumbersome.

4. Permit downward casts in primitive types for parasites

⊕ This solution treats primitive types in the same way as reference types.

⊖ It does not fit the semantics of Java very well (there is no corresponding `instanceof` operator for primitive types). However it is a possible choice for future extensions.

Overriding and inheritance

5. Use invariant result type overriding.

⊕ It is standard Java.

⊖ It is a useless and unjustified restriction and limits the power of parasitic methods (and of standard Java's methods in general).

6. Inherit parasitic methods as if declared locally. (Part of an earlier draft of this proposal.)

⊕ This solution has the advantage of the simplicity. It differs from the one we presented here in that a parasitic method does not parasitize more specific inherited methods but, on the contrary, if these methods are also parasitic it is parasitized by them.

⊖ This solution is less expressive; it is possible to add new parasites in a subclass and to override the existing ones, but the structure of the parasite-host relation is inherited as it is and it cannot be modified. In particular, one cannot define contravariant parasites such as ones that cover a whole set of inherited parasites. This is a flagrant violation of object-oriented principles.

7. Adopt a more powerful typing discipline for contravariant parasites: the host is redeclared with the return type of the parasite (possibly a strict subtype).

⊕ More precise typing discipline that (slightly) enhances the language's expressiveness.

⊖ This extension would have complicated the typing discipline for parasites by requiring additional type rules such as "if a method with a contravariant parasite is overridden, the return type must be a subtype of the contravariant parasite."

8. Parasitic method definitions are not inherited in subclasses.

⊕ It fits the concept of "parasite" according to which a parasite attaches to a particular host.

⊖ It does not fit Java's and, more generally, object-oriented inheritance philosophy according to which a method definition is valid in all subclasses as long as a new definition with the same signature is not given.

Implementation

9. Rewrite runtime system.

⊕ Potentially more efficient.

⊖ This solution may slow down programs *not* using parasitic methods.

⊖ Implementation becomes harder to optimize.

⊖ Makes adoption of multi-methods much less likely.

10. Call parasitic method signature directly, rather than create new `final` methods.

⊕ Potentially more efficient since fewer dispatches are required.

⊕ If a parasite is overridden, then the translation would not need to override the host as well.

⊖ Overridable parasitic bodies make analysis and thus optimization harder.

⊖ Overridable methods are potentially more expensive to call than `final` methods.

⊖ In certain situations, `super` calls leads to surprising infinite recursion.

6 Practice

The practical benefit of parasitic methods is manifest. The simplest example is given by the method `Object.equal` that is used and overridden in a large number of Java programs. Even in the simple built-in class `java.lang.Boolean`, we have:

```
public boolean equals(Object obj) {
   if ((obj != null) && (obj instanceof Boolean)) {
      return value == ((Boolean)obj).booleanValue();
   }
   return false;
}
```

With parasitic methods this can be equivalently (and more naturally) written as

```
public parasitic boolean equals(Boolean obj)  {
  return (obj != null) && (value == obj.booleanValue());
}
```

A much larger example of code in which parasitic methods would be useful is the method `sameStructure` defined in class `UpdatableImpl` of Doug Lea's `collections` package version 0.96.[9]

Moreover, the systematic use of the `parasitic` keyword in every overloaded method definition transforms the overloaded method selection from static into dynamic. In other words, the method to execute is selected according to the dynamic type of its arguments rather than the static ones. Of course the price for this is run-time overhead, but the resulting behavior is, in our opinion, much more intuitive and predictable. For instance, the following program (due to Doug Lea)

```
class Classifier {
  String identify(Object x)  { return "object"; }
  String identify(Integer x) { return "integer"; }
}

class Relay {
  String relay(Object obj)
      { return (new Classifier()).identify(obj); }
}

public class App {
  public static void main(String[] args) {
    Relay relayer = new Relay();
    Integer i = new Integer(17);
    System.out.println(relayer.relay(i));
  }
}
```

prints "`object`" as result, while a definition of `Classifier` such as

```
class Classifier {
  String identify(Object x)  { return "object"; }
  parasitic String identify(Integer x)
  { return "integer"; }
}
```

prints "`integer`".

7 Related Work

Agrawal *et al* [ADL91] describe the basic problem of type-checking systems of multi-methods and forming precedences between applicable method branches. Checking types has two parts: the set of method branches comprising a multi-method must be mutually *consistent*, and the static call sites must type-check. The first part consists of applying the rule that if one method branch is more specific than another, the former's return type must be included in the latter's. In our context, this rule is the one that a parasite's return type must be an assignment subtype of the host's return type. The second part consists of checking the call using the static types of the arguments to compute the static type of the result. In our context, this check is already performed by the Java compiler.

The same condition as consistency was found in the more type-theoretic approach of [CGL92, Cas97], where it was

[9]See `http://gee.cs.oswego.edu/dl/classes/collections` for code and documentation of this package.

called *covariance condition*. There a second condition on the formation of multi-methods was required, namely the existence of a most specialized branch. Such a condition is not necessary (in a sense, it is trivially satisfied) in our approach since the use of textual order makes the search order be total.

Mugridge, *et al.* [MHH91] define the language Kea, where multi-methods and encapsulation are obtained thanks to a mechanism similar to the *partial overriding* of [BC96]. However, the self parameter is considered a parameter like the others (thus this approach is not strictly comparable to ours), and programs may not be modular since the addition of a class can invalidate previously acceptable multi-methods. Furthermore, the approach seems incompatible with Java's overloading and thus not applicable to our specific problem.

Chambers and Leavens [CL95] show that type-checking uses of multi-methods in a powerful language with modules (Cecil) can be done in the context of separate compilation. They also describe an algorithm for statically checking whether a set of multi-method implementations is complete and consistent. However, the test must be performed at link-time, as it requires a list of all concrete classes. Our proposal permits checking to be done separately for each Java class.

8 Future work

Our proposal here is not meant to be the final word on parasitic methods. We have several extensions that we are considering:

1. The set of hosts of a parasitic method is currently determined by the rules of Definition 1. We want to allow the programmer to explicitly define the set of hosts of a parasites by specifying their signatures in an `extends` clause that precedes the body of the parasite. This possibility, besides giving a total control over the parasite-host relation, would also allow every kind of specialization (covariant, contravariant, invariant), at the granularity of parameters. Thus for example

```
class A {
   void m(A x, B y)
          { ... }
}

class B extends A {
   parasitic void m(B x, A y) extends  m(A x, B y)
          { ... }
}
```

the parasite in `B` specializes the method in `A` covariantly on the first parameter and contravariantly on the second.

2. The implementation of the prototype we propose is based on the translation we described in Section 4. Such an implementation is far from being the most efficient one. Many optimizations are possible. For example when a multi-method is called, the most specific method (if it exists) is selected among those that handle the arguments. However this process of selection may involve several dispatchings. For instance, if a programmer declared the methods of a class so that a more specific parasite always precedes the less specific ones, then in the worst case, the message can be dispatched successively to all the parasitic methods; the first dispatches to the second that dispatches to the third and so on. A clever compiler could drastically reduce this cost by calculating the method to be executed and by dispatching directly to it. Other examples of possible optimizations include inlining the final \$-methods into the hosts, and redirecting dispatches to methods with contravariant parasites to go directly to the parasite.

3. We may consider dispatching also on primitive types. A difficulty here is that there is no analog to "instanceof" to determine, for instance, if a `double` value can be represented as an `int`.

9 Conclusion

Multi-methods are selected dynamically based on the type of arguments, and provide a type-safe solution to the problem of binary methods with inheritance. Our implementation of them in Java, parasitic methods, unifies covariant and contravariant specialization in a single feature. Parasitic methods give the programmer run-time multi-method dispatch in the same spirit as Java's current (static) method selection. They permit the programmer to write more elegant and readable code than the equivalent (but error prone) use of `instanceof` and type casts.

The extension preserves Java's type-safety, modularity and separate compilability, and has a cost only when it is used. In fact, inasmuch as the existence of multi-methods encourages programmers to write specialized efficient versions of methods, programs using parasitic methods may be *faster* than ones without. Furthermore, we have learned that some instructors recommend students never use type overloading in Java methods (at least when they have related parameter types), because static resolution yields unexpected behavior. Parasitic methods would make type overloading acceptable again.

Parasitic methods are straightforward to implement; we currently have an implementation on top of the Java Development Kit (version 1.0.2).

Parasitic methods would work nicely in Java extended with parametric polymorphism as proposed by Odersky and Wadler [OW97] and by Banks *et al* [BLM97]. Both encapsulated multi-methods and parametric classes are features originally proposed in the type theory community. We are pleased to see that they are starting to take root in more practical settings.

Finally most design trade-offs in our work derive from the fact that we wanted to define a conservative extension of Java. Most of the difficulties in particular were due to keeping the semantics of Java's static overloading unchanged. Indeed a cleaner, but non-conservative, extension would be to consider all methods as parasitic methods (this is what it is done in KOOL [Cas97] and what we proposed [BC96] for the database programming language O_2). Practically speaking, this solution consists precisely in having dynamic (rather than static) resolution for Java overloaded methods. In any case, we are certain this would be the best solution for a brand new language.

Acknowledgments

The authors are grateful to Manuel Fähndrich, Edwin Chan, and Doug Lea for their valuable information and suggestions. We are sincerely obliged to the OOPSLA's referees whose pertinent remarks stimulated us to deeply modify and, we hope, decisively improve the work we had submitted. Special thanks to Gary Leavens for his constructive remarks on several drafts.

References

[ADL91] Rakesh Agrawal, Linda G. DeMichiel, and Bruce G. Lindsay. Static type checking of multi-methods. In *Conference Proceedings of OOPSLA'91 – Object Oriented Programming Systems, Languages and Applications*, pages 113–128, New York, October 1991. ACM Press. Appeared as *ACM SIGPLAN Notices 26* (11), November 1991.

[BC96] John Tang Boyland and Giuseppe Castagna. Type-safe compilation of covariant specialization: A practical case. In *ECOOP '96 — Object-Oriented Programming (10th European Conference)*, volume 1098 of *Lecture Notes in Computer Science*, pages 3–25. Springer-Verlag, Berlin, Heidelberg, New York, July 1996.

[BCC$^+$96] K. Bruce, L. Cardelli, G. Castagna, The Hopkins Object Group, G. Leavens, and B. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1996.

[BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Implementing an Object-Oriented Database System: The Story of O_2*. Morgan Kaufmann, 1992.

[BLM97] Joseph A. Banks, Barbara Liskov, and Andrew C. Meyers. Parameterized types and Java. In *Conference Record of the Twenty-fourth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 132–145, New York, January 1997. ACM Press.

[Cas97] G. Castagna. *Object-Oriented Programming: A Unified Foundation.* Progress in Theoretical Computer Science series. Birkäuser, Boston, 1997.

[CGL92] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. In *ACM Conference on LISP and Functional Programming*, pages 182–192, 1992. Extended and revised version in *Information and Computation* 117(1):115-135, 1995.

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

[DE97] Sophia Drossopoulou and Susan Eisenbach. Java is type safe – probably. In *ECOOP '97 — Object-Oriented Programming (11th European Conference)*, volume 1241 of *Lecture Notes in Computer Science*, pages 389–418. Springer Verlag, Berlin, July 1997.

[DG87] L. G. DeMichiel and R. P. Gabriel. Common Lisp Object System overview. In Bézivin, Hullot, Cointe, and Lieberman, editors, *Proc. of ECOOP '87 European Conference on Object-Oriented Programming*, number 276 in Lecture Notes in Computer Science, pages 151–170, Paris, France, June 1987. Springer.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, Reading, Massachusetts, 1996.

[MHH91] W. B. Mugridge, J. Hamer, and J. G. Hosking. Multi-methods in a statically-typed programming language. In Pierre America, editor, *Proceedings of ECOOP '91 Conference, Geneva, Switzerland*, volume 512 of *Lecture Notes in Computer Science.* Springer, 1991.

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

[Ste90] Guy Steele. *Common Lisp the Language.* Digital Press, 2nd edition, 1990.

[Str94] Bjarne Stroustrup. *The design and evolution of* C++. Addison-Wesley, Reading, Massachusetts, 1994.