Type-Safety and Overloading in Sather

B. Gomes, D. Stoutamire and B. Weissman

TR-97-055

December 1997

Abstract

Method overloading is a form of statically resolved multi-methods which may be used to express specialization in a type hierarchy[GSWF97]. The design of the overloading rule in Sather is constrained by the presence of multiple-subtyping, and the ability to add supertyping edges to the type graph after-the-fact [SO96]. We describe the design of overloading rules which permit method specialization while allowing separate type-checking i.e. existing code cannot be broken by after-the-fact addition of supertyping edges.

1.0 Introduction

Method overloading is used in singly dispatched object-oriented languages to achieve a form of statically resolved, type-safe multi-methods. Overloading permits interfaces and to support more than one method of the same name but with declared arguments of different types - calls to the method are resolved based on the types of the arguments in the call. In the course of the design of the Sather libraries, it was discovered that overloading was needed to support type-safe specialization [GSWF97], a point whose theoretical underpinnings were described in [Cas95].

Sather [Omo95] originally supported a minimal form of method overloading - two methods of the same name were permitted to coexist in an interface if they had:

- different numbers of arguments, or
- differred in the presence/absence of a return type

We will take these two trivial forms of overloading for granted, and will confine our attention to considering methods with the same numbers of arguments/return value. The original Sather also supported a limited form of overloading based on monomorphic types, which is subsumed by the more general rules described here.

We start by making a case for the two features of Sather that make the overloading rule described necessary, namely parametric polymorphism based on subtyping constraints. We argue that these are desirable features in an object-oriented language and have advantages over the alternatives. We describe the goals of our overloading rule, and how permitting general overloading violates these goals. We then describe the chosen overloading resolution rule, and some alternatives which might also be acceptable.

1.1 Constrained Parametric Polymorphism

Most modern object oriented languages permit a form of parametric polymorphism, in which a parametrized class (sometimes called a generic or template class) may be described using one or more type parameters. When the class is actually used, these type parameters must be instantiated with actual ty[pes. In order to permit the separate type-checking and compilation of parametrized classes, it is necessary to constrain the type bounds [DGLM95]. There are two common methods used to constrain the type bounds:

- Implicit conformance bounds, as with the the where clauses of Theta [DGLM95], the type maching clauses of Emerald [JLHB88] and the property classes of Sather-K [Goo97]. In this case, the type bound states the signatures of the methods that must be provided by any instantiating type. However, the type bound itself is not a type, and is outside the type system. Hence, the type parameter may be instantiated by any class that supports the methods specified in the type bound.

- Explicit subtyping bounds, as in Sather and Rapide [KLMM94]. Here, the type bound on the parameter refers to an actual type; all instantiations of the parameter must be subtypes of the typebound.

```
With conformance:
class VECTOR{T}
   where T is plus(T):T;  minus(T):T; ... etc;

With explicit subtyping:
class VECTOR{T < CPX}
```

There are pros and cons to both choices; some of the deficiencies of explicit subtyping are described in [DGLM95]. However, there are advantages to explicit subtyping as well:

- Subtyping bounds make it possible to constrain a parameter based on the semantics of the methods involved (provided subtyping relationships preserve class invariants), rather than on the (potentially co-incidental) naming of methods.
- Contrary to the claims in [DGLM95], we have found in the design of the Sather libraries that subtyping bounds are frequently (large) regular classes and may well be used in contexts other than the type bounds. For instance, in an algorithm class that deals with vectors and matrices, it is quite reasonable to have the whole matrix and vector abstractions as type bounds. Forcing the use of a separate mechanim to express these constraints essentially requires duplicating class interfaces for the sake of type bounds.

The implicit conformance approach may be seen as a variant of the subtyping approach in which all possible subtyping edges between type bounds and actual classes are present and cannot be avoided. By contrast, the explicit subtyping approach allows the programmer to specify exactly which subtyping edges are permissible, thus providing the programmer with the ability to more tightly control (and thereby better ensure the safety and correctness of) the behavior of a parametrized class.

1.2 Supertyping

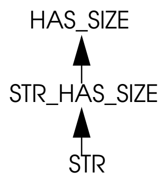
A problem with using explicit subtyping to specify parameter constraints arises when using code that may not be modified. For the sake of simplicity, we will refer to an (arbitrary) body of stand-alone, unmodifiable code as a “library”. Consider writing a parametrized class that makes use of the “size” method of a type parameter.

```
abstract class HAS_SIZE is
   size:INT;
end;
class BAR_ALG{T < HAS_SIZE} is
   foo(a:T) is  i:INT := a.size;...  end;
end;
```

Suppose we now wished to parametrize BAR_ALG using an library class which supports the size method, such as STR. Since the user cannot change the library directly, it is not possible to have STR subtype from HAS_SIZE.

To get around this problem, Sather permits the user to add subtyping edges, where valid, after the fact. A supertyping clause achieves this effect

```
abstract class STR_HAS_SIZE
    subtype of  HAS_SIZE
   supertype of  STR;
```



The only edge additions permitted are those that are guaranteed to be type-safe i.e. the parent and child that are related by the supertyping clause must be already contravariantly conformant. If this conformance relationship does not hold, the supertyping statement is not permitted.

By contrast, the implicit conformance based approach may be viewed as automatically adding all possible supertying edges to the type graph whenever a type bound is encountered.

1.3 Call SIgnatures

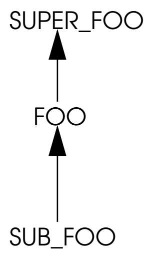
Since this report deals mainly with the resolution of calls, and there are many different call signatures involved, we define our terminology as follows:

Declared signature or Interface signature. This is the signature of the method as it is declared in an interface. We refer to the types of the arguments in the interface signature as the declared argument types.

Call signature. This method signature is determined by the declared types of the arguments in a call to a method.

Execution signature. This signature is determined by the types of the actual object used in the call. The dynamic signature of a method is not available at compile time.

```
class EXAMPLE is
   bar(arg:SUPER_FOO);
sub_foo:SUB_FOO;
foo:FOO := sub_foo;
-- The declared type of foo is FOO
--  the dynamic type is SUB_FOO
example:EXAMPLE;
example.bar(foo);
```



The dynamic signature of a method is not available at compile time and is not relevant to the resolution of overloading. Overloading resolution, in general, involves the matching of a call signature to one of many interface signatures. In the above piece of code, consider the signature of example.bar(foo)

- The declared signature is EXAMPLE::bar(SUPER_FOO)
- The call signature is EXAMPLE::bar(FOO)
- The execution signature is EXAMPLE::bar(SUB_FOO)

2.0 Goals

A general goal of the Sather type system is that type safety must be statically guaranteed and locally checkable. By local checkability we mean that if any module of code is type-checked, its typesafety cannot be violated when it is combined with other, similiarly type-checked, code (modulo trivial class-level naming conflicts, since Sather has no module mechanism).

The design of the Sather overloading rule was mainly driven by the need to support consistent specialization, as described in [GSWF97]. It was essential that a method A be able to coexist with a B, if B is a specialization of method A. Method B is a specialization of method A if the precondition of method B implies the precondition of method A [FNZ97], and the postcondition of method A imply the postcondition of method B. In terms of argument types - B is a specialization of A if its argument types are subtypes of the argument types of A. Thus, specialization may be seen to be the converse of the usual conformant rule for substitutability.

We present a brief example that illustrates the kind of overloading we wish to support

```
class CPX
   plus(arg:CPX):CPX;
class REAL subtype of CPX
   plus(arg:CPX):CPX;
   plus(arg:REAL):REAL;    -- Specialization of plus(arg:CPX):CPX
```

We wish to support the specialization relation (real numbers are a specialized form of complex numbers) between a subtype and a supertype. Specialization requires the support of covariant overloading, where the argument types in the specialized method are subtypes (i.e. specialized versions) of the arguments in the more general method.

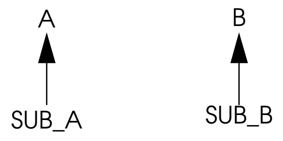
3.0 The Problem

The straightforward solution is to permit overloading based on any argument types, as is done in C++ and Java. Calls are then resolved by requiring that there be a single, most specific method that matches the call signature - ambiguity is prohibited. This approach will not work in a language such as Sather, which permits type-safe supertyping. The addition of such supertyping edges may cause existing code to break (method overloading in existing code that was unambiguous may become ambiguous due to the addition of such supertyping edges).

3.1 Supertyping may breaking existing overloaded calls

To illustrate the problem, consider the following example.   :

```
class FOO  is
   foo(a:A);
   foo(b:B);
```

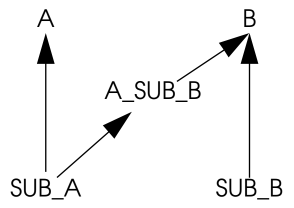


The following calls on FOO may be unambigously resolved:

```
f:FOO
sub_a:SUB_A;
sub_b:SUB_B;
f.foo(sub_a);      -- overloading resolves to call FOO::foo(A);
f.foo(sub_b);      -- overloading resolves to call FOO::foo(B)
```

Having assured ourselves of the typesafety of the above code, we now add a supertyping clause that makes SUB_A a subtype of B as well (this is only permitted if the interface of SUB_A actually conforms to the interface of B):

```
class A_SUB_B
   supertype of B
   subtype of SUB_A
```



The addition of this supertyping edge, causes our original piece of code to break. The second overloaded call (with argument of type SUB_B) still has a single most specific method, but the first overloaded call with argument of type SUB_A is now ambiguous.

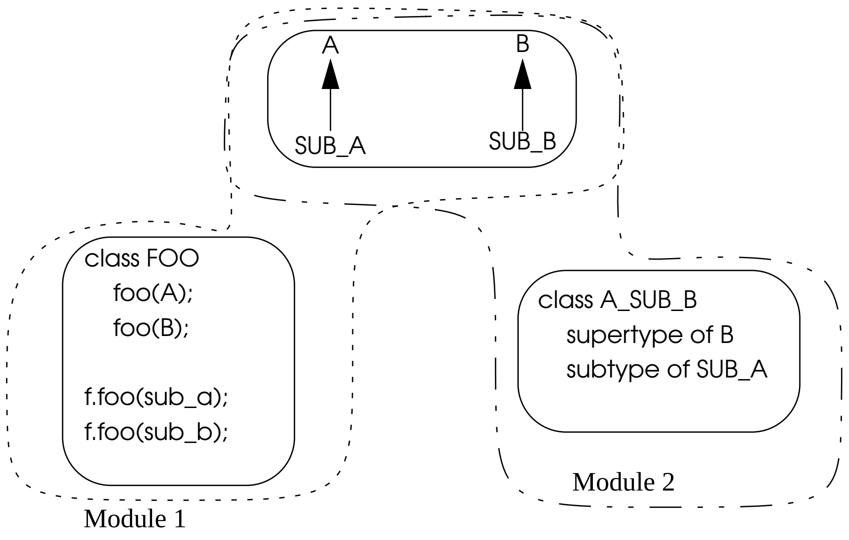
```
f:FOO
sub_a:SUB_A;
sub_b:SUB_B;
f.foo(sub_a);       -- overloading resolves to call FOO::foo(A);
f.foo(sub_b);      -- overloading is ambiuous resolves to call FOO::foo(B)
```

The real problem is not that the overloading resolution is ambiguous, but rather that, after code has been deemed correct, typing errors may be introduced by unrelated code. This renders type-checking a non-local problem, and destroys the compositionality of code.

3.2 Supertyping may violate separate type-checking of overloaded

calls

Another way to look at the above problem is that compositionality is lost; errors may occur when two pieces of code are combined, even though each was individually typesafe.



In the diagram above, both module 1 and module 2 will type check independantly (the definitions of the classes A and B arise from some common source, such as the general libraries). However, when module 1 and module 2 are combined, the overloading resolution of module 1 is broken - the call foo(sub_a) is ambiguous and therefore forbidden.

3.3 Restricting Supertyping

There are several possible solutions to the above dilemma that involve restricting supertyping.

A drastic solution is to eliminate supertyping from the language. However, as we havepointed out in Section 1.2, supertyping is necessary for reuse of unmodifiable library code when parametrized type constraints are based on explicit subtyping.

A less drastic solution is to permit either a supertyping or a subtyping clause in a class, but not both. This will eliminate the problems described in the previous section, but also prohibits useful subtyping relationships. Consider acquiring two separate libraries, for instance a library of parametrized container classes and a library of classes which we may wish to insert into the containers. Inserting objects into the containers requires placing them under the type-bounds specified in the container classes; this may be achieved by an intermediate classes that are under the type bound and over the contained classes.

We have taken the approach of determining judicious limitations on the kinds of overloading permitted, rather than limiting supertyping. Note that overloading is statically resolved and a functionally equivalent effect may be obtained by distinguishing between method names that would otherwise be overloaded. Supertyping, however, has deeper semantic consequences, making it harder to work around limitations.

4.0 The Overloading Rule in Sather

Instead of restricting supertyping, we restrict the overloading rule to permit exactly the overloading required for specialization.

4.1 Conflict and Conformance

Permissible overloading in an interface is constrained by the rule for *conflict*, which defines when a method f and g may not coexist in an interface. We assume that both methods have the same number of arguments, and that both either have a return value or neither does[1].

> Method signature **f conflicts with g** when each argument type in f is neither a subtype nor a supertype of the corresponding argument type in g.

The other half of the overloading rule is matching a call signature to a particular interface signature. There must exist a method in the interface with the specified method name such that for each argument, the type of each expression in the call is a subtype of the declared type of the corresponding argument. If there is more than one such method, there must be a **unique one which is most specific, conforming to all the others**. This definition is completed by the definition of conformance, which is the standard contravariant-argument, covariant return type rule.

> Method signature **f conforms to g** when for every argument, the type of the argument in g is a subtype of the corresponding argument type in f; and if it has one, the return type of f is a subtype of the return type of g.

In other words, the rule for conflict ensures a total order on each argument position in overloaded methods that coexist. The rule for conformance demands a total order on all matching methods (which is more restrictive than just an order on each argument position).

4.2 Why does this solve the problem?

We first note that all the problems with permitting overloading based on any difference in arguments arose because two argument types that were initially unrelated, were later made related through an edge introduced by supertyping. The above definition forces the corresponding arguments of any two methods in the interface to always have a subtyping relationship. As a consequence, any addition of supertyping edges between argument

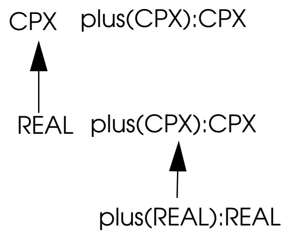
1. These rules are from the specification [SO96], minus details regarding the mode of the arguments and special cases when the argument types happen to be concrete.

types after-the-fact cannot change the existing subtyping relationship between the arguments

```
class CPX
   plus(arg:CPX):CPX;

class REAL subtype of CPX
   plus(arg:CPX):CPX;
   plus(arg:REAL):REAL;

class COMP_NUMBER
   supertype of REAL;
   subtype of CPX;
```



More formally, we wish to show that the later addition of supertyping edges to the type graph can never change the resolution of a call to an overloaded method.

We start with a call on an overloaded method has been unambiguously resolved with a single method whose declared signature has the most specific match to the call signature in every argument position. We proceed by showing that the most specific match in each argument position cannot be changed by the addition of supertyping edges. We proceed by showing that the rule for conflict ensures a total order between the argument types in each argument position which cannot be affected by the addition of supertyping edges; the subtyping ordering between the types in each argument position cannot change, then neither can the choice of most specific method.

To show that our rule for conflict ensures a total order on the types in every argument position, consider a set of overloded methods:

Consider the set of types, *S*, that occur in a particular argument position among all the overloaded methods. The conflict rule enforces a subtyping relationship, or partial order, between any pair of types in *S*. Since there is a partial order between every pair of types in *S*, there is a total order on *S*.

Assume that the addition of a supertyping edge introduces a new relationship between two types in *S*. That is, for a pair of types *S1* and *S2*, where initially $S1 < S2$, after introducing a supertyping edge, $S2 < S1$. But the original typing relationship between *S1* and *S2* must still hold (typing edges can never be deleted from the type graph). Hence, since $S1 < S2$ and $S2 < S1$, the supertyping edge has introduced a cycle into the type graph. Defining a class which introduces a cycle into the existing type-graph is prohibited. Therefore we have a contradiction. Therefore, we cannot add a supertyping edge that introduces a new relationship between the types in *S.*

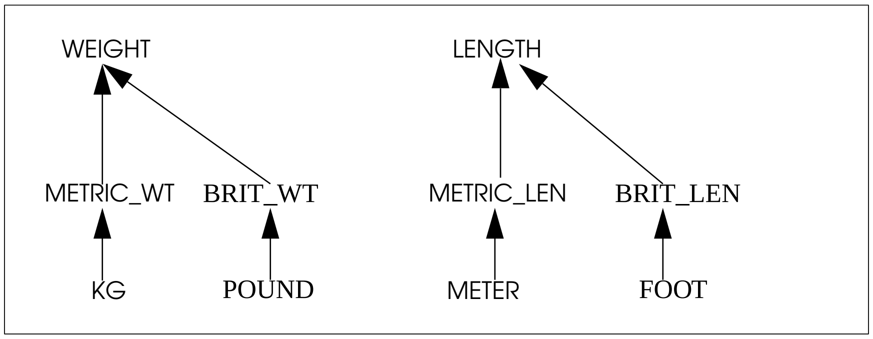
Hence, there is a total order on the types in every argument position of the overloaded methods which cannot be changed by any later addition of supertyping edges.

4.3 Why not also use the return type to resolve conflicts?

According to the current overloading rules, the type of the return value and out arguments cannot be used to differentiate between methods in the interface. There is no theoretical reason to disallow this possibility. However, permitting overloading based on such return values involved additional implementation work and was not required for the usages we envisaged (the interface signature with the most general return type must be chosen, in addition to picking the interface sigature with the most specific arguments). Thus, overloading is not permitted based on differences in the return type (or out arguments, which are equivalent to return types) of a method

5.0 Understanding Overloading Resolution

To clarify our overloading rules, we can visualize the overloaded resolution with the aid of a simple example of special units of weight. These units of weight can provide methods such as addition to other units of height and weight.]



It is easy to see that the above hierarchy may be extended with other units of weight such as grams and ounces, and by units of height such as inches. Weights and heights may be used in a method which determines the postage for a package

```
(1)postage(weight:KG, maximum_side:METER):DOLLARS;
```

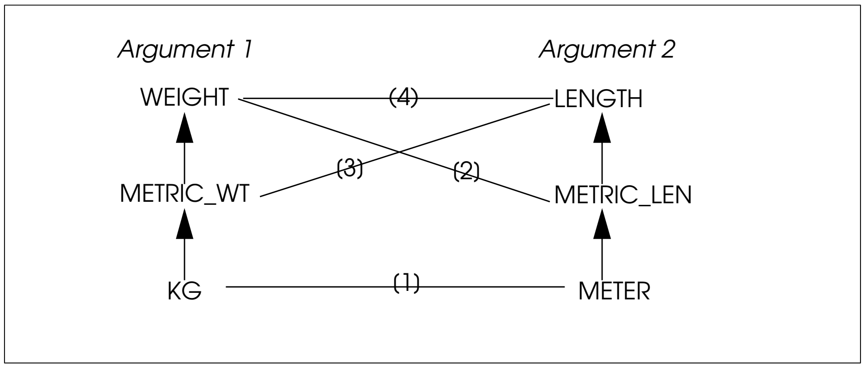
Though the postage may be directly computed using kilograms and meters, it is possible to compute the postage in the more general case by performing either converting the weight or the length from other units, at some cost in efficiency[1].

```
(2)postage(weight:WEIGHT,length:METRIC_LEN):DOLLARS;
(3)postage(weight:METRIC_WT,length:LENGTH):DOLLARS;
(4)postage(weight:WEIGHT,length:LENGTH):DOLLARS;
```

The above methods do not conflict, and there is therefore a total order between the argument types in each argument position. If we start out by

1. Admittedly, efficiency is an odd concern in this small example. However, efficiency may be of significant concern in other real library classes such as matrices and vectors, where conversion between different types can be expensive.

considering the total order in each argument position, we may visualize the above set of methods as shown below.



Consider the following variables

```
meter:METER := METER::create(1.0);
kg:KG := KG::create(1.0);
weight:WEIGHT := kg;
length:LENGTH :=  meter;
metric_len:METRIC_LEN := foot;
metric_wt:METRIC_WT := kg;
```

These variables may be used in overloaded calls as shown below

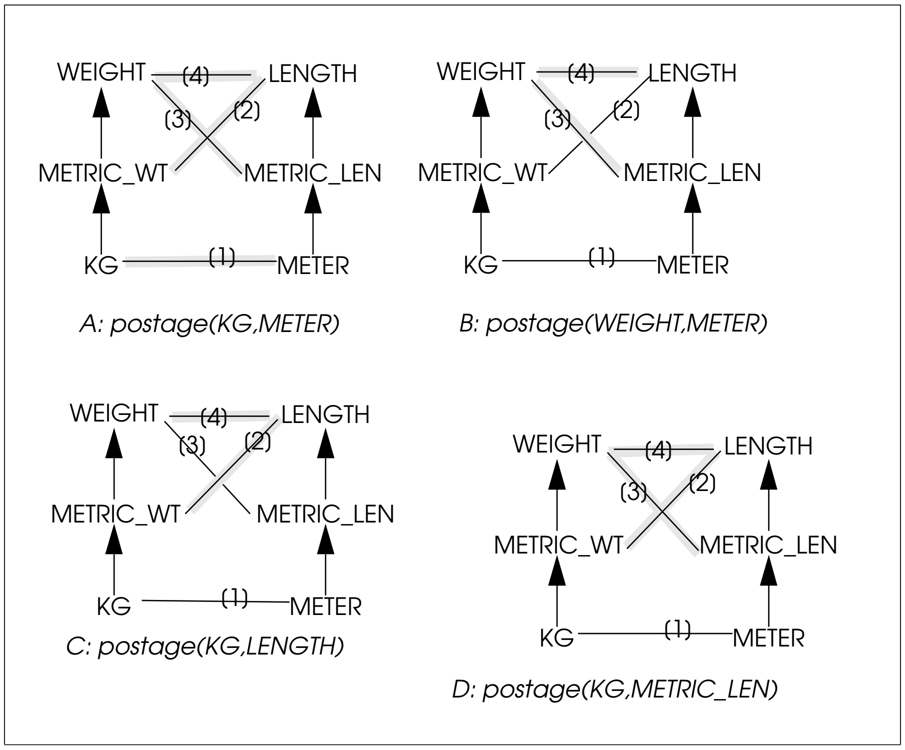
	Call	**Call Signature**	**Most Specific Match**
A	(kg,meter)	(KG,METER)	(KG,METER)
B	(weight,meter)	(WEIGHT,METER)	(WEIGHT,METRIC_LEN)
C	(kg,length)	(KG,LENGTH)	(METRIC_WT,LENGTH)
D	(kg,metric_len)	(KG,METRIC_LEN)	(METRIC_WT,LENGTH) (WEIGHT,METRIC_LEN)

The first three of the above calls can be resolved unambiguously, by choosing the most specific method. In the diagram below, the matching methods for each call are highlighted. In all cases except D, there is single unambiguously lowest match. In D, however, there are two matching methods (2) and (3), neither of which conforms to the other. The ambiguity in D can be resolved if the programmer upcasts one of the argument types to a more general type (for instance, by assigning wt:WEIGHT := kg and then performing the call using wt).

6.0 An Alternative: Preventing Ambiguity at the Point of Call

The rule we have described so far meets all our design criteria. However, it is possible to run into situations where ambiguities in resolving an overloaded call occur. These ambiguities may be avoided if we change the definition of conflict in our overloading rule.

> Two methods f and g conflict in an interface iff neither f conforms to g nor g conforms to f.



With this new definition of conflict, which forces a conformance relationship between every pair of overloaded methods, we have a total order on all the overloaded methods. Overloading resolution at the point of call now amounts to picking the most specific method, rather than the a set of methods which are most specific in each argument position. The ambiguity mentioned in the previous section cannot now occur, since the declarations of (2)postage and (3)postage (3) would conflict. This second choice actually permits exponentially fewer overloaded methods than then previously mentioned overloading rule.

There are trade-offs between these two choices:

Our original choice of overloading permits certain, potentially useful, cases of overloading to occur. Furthermore, any conflict at the point of call can always be resolved by a suitable upcasting of one or more of the call argument types.

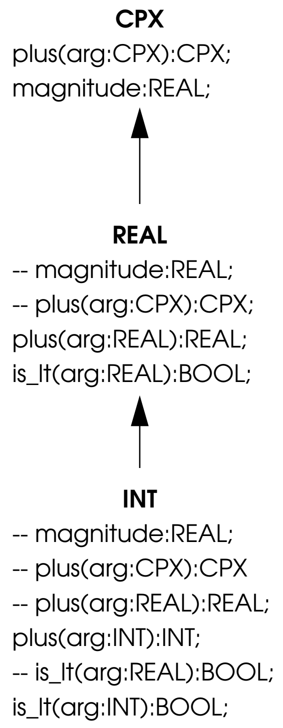
The overloading rule presented in this section results in no overloading conflict errors at the point of call, essentially prohibiting all such conflicts at the point of definition of the interface. These errors of conflict may be somewhat hard for the user of a class to understand.

Other variants of the rule are also possible, such as using the lexicographic order of the arguments to decide disambiguation.

For the rest of this report, we will confine our attention to the overloading rule presented in Section 5.0, since that is the rule that was adopted and implemented by Sather, though all the choices mentioned above would be quite reasonable.

7.0 An Example

The above overloading rule provides a clean implementation of the standard hierarchy of numerical entities.



```
CPX
plus(arg:CPX):CPX;
magnitude:REAL;

REAL
-- magnitude:REAL;
-- plus(arg:CPX):CPX;
plus(arg:REAL):REAL;
is_lt(arg:REAL):BOOL;

INT
-- magnitude:REAL;
-- plus(arg:CPX):CPX
-- plus(arg:REAL):REAL;
plus(arg:INT):INT;
-- is_lt(arg:REAL):BOOL;
is_lt(arg:INT):BOOL;
```

The commented out methods are obtained by subtyping and need not be explicitly mentioned (though mentioning them in the subtype is not prohibited). Using this type structure and the overloading rule from Section 5.0, we can perform additions within a particular domain, say the domain of real numbers.

```
a:REAL;
 b:REAL;
c:REAL := a.plus( b);    -- chooses REAL::plus(REAL):REAL
```

The methods plus(CPX):CPX and plus(REAL):REAL both match, and the latter method is more specific. However, operations between real and complex numbers are also permitted

```
a:REAL;
c:CPX;
d:CPX  := a.plus(c);   -- chooses REAL::plus(CPX):CPX
```

In this case, only REAL::plus(CPX):CPX matches the call, and is chosen. In the process of the addition, the real number is viewed as a complex number with a zero imaginary part.

8.0 Overloading Resolution in Parametrized Classes

Within parametrized classes, the types of arguments within a signature are sometimes determined by the types of class parameters. A method in a parametrized class has arguments whose types are type parameters. In order to permit the separate typechecking and compilation, the overloading resolution of such methods is done with respect to the type bounds, rather than the instantiated types.

9.0 Other Language Interactions

We have omitted any discussion of how other language features enter into the overloading rule. The more complete overloading rule for Sather is presented in Appendix A. While these interactions are important in Sather, they do not add any extra complexity to the rule.

Classes which are guaranteed to be leaf classes (implementation classes in Sather, final classes in Java) may be safely used while overloading, as permitted in the old overloading rule. A method with an implementation class as a declared argument type may only be used when there is an exact match between the call argument type and the declared argument type. No later addition of supertyping edges can change this exact match. In practice, overloading based on implementation types is extremely common and useful in Sather code.

Argument modes also interact with the issue of overloading. In general, overloading is not permitted based on out or inout arguments, since they behave like return values. Overloading is only permitted based on once and in arguments. However, the presence or absence of out or inout mode specifiers may be used to permit overloading.

10.0 Implementation

The overloading rule, as describe here, has been implemented in the current release of the Sather compiler. It has been successfully used in the current (internal) Sather libraries in the design of the container abstractions. A rewriting of the Sather container, graph and math libraries is currently underway, which will more fully exploit the potential for specialization as supported by the overloading rule.

11.0 Conclusions

Sather supports subtyping based parameteric polymorphism, in conjunction with supertyping. An overly permissive overloading rule would result in an inability to type check code separately. We describe a design in which overloading for specialization is supported, while still maintaining compositional type-checking in the presence of supertyping. We also describe an alternative, more restrictive, overloading rule in which ambiguities cannot arise at the point of call. The overloading rule described in Section 5.0 has been implemented in the Sather compiler, and is in use in the Sather libraries.

Appendix A The Specification of Overloading

For the sake of convenience, we gather here the various portions of the Sather specification [SO96] that, taken together, define the overloading rules for Sather. The definitions below take into account other complexities of the overloading rule that arise from the presence of argument modes and leaf (implementation or concrete) classes in the type graph.

11.1 Signatures (section 2.3.2)

We say that the method signature **f conflicts with g** when:

- f and g have the same name and number of arguments,
- f and g either both return a value or neither does,
- each argument mode in f is the same as the corresponding mode in g, or the mode in
  one is 'in' while the other is 'once',
- and each argument type in f is neither a subtype nor a supertype of the corresponding
  argument type in g, unless both are concrete.

This rule for signature conflict defines which methods may be overloaded. Sather permits overloading based on the number, type and mode of arguments, as well as whether or not a return value is present. However, overloading is not permitted between 'in' and 'once' modes.

We say that the method signature **f conforms to g** when

- f and g have the same name and number of arguments,
- f and g either both return a value or neither does,
- the mode of each argument is the same (in, out, inout or once),
- contravariant conformance:
  for any 'in' or 'once' arguments, the type in g is a subtype of the type in f;
  for any 'inout' arguments, the type in f is the same type as in g;
  for any 'out' arguments, the type in f is a subtype of the type in g; and
  if it has one, the return type of f is a subtype of the return type of g.

11.2 Method call expressions (section 2.7.3c)

Sather supports routine and iterator overloading. In addition to the name, the number,types, and modes of arguments in a call and whether a return value is used all contribute to the selection of the method. The modal_list portion of a call must supply an expression corresponding to each declared argument of the method. There must exist a method with the specified name such that:

- for each 'in' and 'once' argument, the type of each expression is a subtype of the declared type of the corresponding argument, and
- for each 'out' argument, the type of each expression is a supertype of the correspond
  ing argument, and
- for each 'inout' argument, the type of each expression is the exact type of the corresponding argument.

If there is more than one such method, there must be a **unique one which is most specific, conforming to all others**.

11.3 Overloading in parametrized classes (section 2.7.3c)

When argument expressions have the type of a class parameter, the type constraint of that parameter is used to select the most specific method, rather than the realized type of the parameter. Overloading may not occur solely by the type of out arguments or return type; there must be at least one non-out argument of differing type between the most specific method and any others.

11.4 void expressions (section 2.7.4)

[void expressions may be used] as an argument value in a method call or in a creation expression. In this last case, the argument is ignored in resolving overloading

References

[Cas95] Guiseppe Castagna. **Covariance and contravariance: Conflict without a cause.** *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, March 1995.

[DGLM95] Mark Day, Robert Gruber, Barbara Liskov, and Andrew Myers. **Subtyping vs. where clauses: Constraining parametric polymorophism.** In *OOPSLA95*, 1995.

[FNZ97] Jozsef Frigo, Rainer Neumann, and Wolf Zimmermann. **Mechanical generation of robust class hierarchies.** In *TOOLS97*, 1997.

[Gom97] Benedict Gomes. **A proposal for safe covariant type specifiers in abstract types.** Technical Report Unknown, International Computer Science Institute, July 1997.

[Goo97] Gerhard Goos. **Sather-k, the language.** *Software - Concepts and Tools*, 1997. To appear.

[GSWF97] Benedict Gomes, David Stoutamire, Boris Weissman, and Jerome Feldman. **Using value semantic abstractions to guide strongly typed library design.** Technical Report Unknown, International Computer Science Institute, July 1997.

[JLHB88] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. **Fine-grained mobility in the Emerald system.** *ACM Transactions on Computer Systems*, 6(1), February 1988.

[KLMM94] Dinesh Katiyar, David Luckham, John Mitchell, and Sigurd Melda. **Polymorphism and subtyping in interfaces.** *ACM SIGPLAN Notices*, 29(9):22–34, Aug 1994.

[Omo95] Steven Omohundro. **The Sather 1.0 specification.** Technical Report TR-95-057, The International Computer Science Institue, 1995.

[SO96] David Stoutamire and Steven Omohundro. **The Sather 1.1 specification.** Technical Report TR-96-012, The International Computer Science Institue, 1996.