Engineering a Programming Language: The Type and Class System of Sather

Clemens Szypersky* Stephen Omohundro†

Stephan Murer‡

TR-93-064

November 1993

Abstract

Sather 1.0 is a programming language whose design has resulted from the interplay of many criteria. It attempts to support a powerful object-oriented paradigm without sacrificing either the computational performance of traditional procedural languages or support for safety and correctness checking. Much of the engineering effort went into the design of the class and type system. This paper describes some of these design decisions and relates them to approaches taken in other languages. We particularly focus on issues surrounding inheritance and subtyping and the decision to explicitly separate them in Sather.

*ICSI, E-mail: `szyper@icsi.berkeley.edu`.

†ICSI, E-mail: `om@icsi.berkeley.edu`.

‡ICSI and Eidgenössische Technische Hochschule (ETH), Zürich, Switzerland. E-mail: `murer@icsi.berkeley.edu`.

1 Introduction

Sather is an object-oriented language developed at the International Computer Science Institute [22]. It has a clean and simple syntax, parameterized classes, object-oriented dispatch, statically-checkable strong typing, multiple subtyping, multiple code inheritance, and garbage collection. It is especially aimed at complex, performance-critical applications. Such applications are in need of both reusable components and high computational efficiency.

Sather was initially based on Eiffel and was developed to correct the poor computational performance of the Eiffel implementations available in 1990. Eiffel introduced a number of important ideas but also made certain design decisions which compromised efficiency. Sather attempts to support a powerful object-oriented paradigm without sacrificing either the computational performance of traditional procedural languages or support for safety and correctness checking.

The initial "0.1" release of the compiler, debugger, class library, and development environment were made available by anonymous FTP[1] in May, 1991 and it was quickly retrieved by several hundred sites. This version achieved our desired efficiency goals [15] and was used for several projects. Our experience with it and feedback from other users has led to the design of Sather 1.0. This improves certain aspects of the initial version and incorporates a number of new language constructs.

The language design process has been intimately coupled with the design and implementation of libraries and applications. A particularly demanding application is the extensible ICSI connectionist network simulator: ICSIM [24]. The examples in this paper are taken from the actual code and structure of the Sather libraries and applications to make them realistic. The design effort was continually a balance between the needs of applications and constraints on language design, such as simplicity and orthogonality.

One of the most fundamental aspects of the Sather 1.0 design is its type system. Earlier versions of the language were strongly typed, but it was not possible to *statically check* a system for type correctness. Eiffel has the same problem [5], and attempts to solve it by introducing *system-level type-checking* [17]. This is a conservative system-wide global check. A system which satisfies the check will be type safe but many legal programs are rejected. Adding new classes to a system can cause previously correct code to become incorrect.

Sather 1.0 solved this and other problems by completely redesigning the type and class system. This paper describes a number of these issues. We do not describe the whole language here but do include the relevant parts of the grammar in Appendix A. The bulk of the paper is devoted to the interplay between subtyping and subclassing. Section 2 defines these concepts and motivates the decision to explicitly separate them in the language. It describes the Sather version of parameterized classes and object-oriented dispatch. It also describes the three kinds of Sather objects: reference objects, value objects, and bound objects. Bound objects are a particularly clean way of implementing higher-order functions within an object-oriented context. Section 3 describes some of the subtle issues involved in code inheritance. Finally, section 4 describes some more system level issues.

2 Sather Types and Classes

Object-oriented terminology is used in a variety of ways in the programming language literature. A few informal definitions will suffice for the purposes of this paper:

- *Objects* are the building blocks of all Sather data structures. Objects both encapsulate state and support a specified set of operations.
- A *type* represents a set of objects.
- The *signature* of an operation that may be performed on an object consists of its name, a possibly empty tuple of its argument types, and an optional return type. Sather supports both *routines* which perform a single operation and *iters* [20] which encapsulate iteration abstractions[2].

[1] From ftp.icsi.berkeley.edu, directory pub/sather/

[2] There is some additional information in signatures which are associated with iters which we do not describe here.

- Each type has an *interface* which consists of the signatures of the operations that may be applied to objects of the type. Sather supports *overloading* which means that an interface may have more than one operation with the same name if they differ in the number or types of arguments or in the presence of a return type.
- *Classes* are textual units that define the interface and implementation of types.
- The Sather *type graph* is a directed acyclic graph whose vertices are types and whose edges define the *subtype* relationship between them. We say that a type $\mathcal{A}$ *conforms* to a type $\mathcal{B}$ if there is a directed path from $\mathcal{A}$ to $\mathcal{B}$ in the type graph.

2.1 Subtyping and Multiple Subtyping

Every object and every variable in Sather has a uniquely specified type. The fundamental Sather typing rule is: "*A variable can hold an object only if the object's type conforms to the variable's declared type.*".

There are three kinds of object type: *reference*, *value*, and *bound*. We describe these later. Variables can be declared by one of these types, but may also be declared by an *abstract* type. These are types which represent sets of object types and are how Sather describes polymorphism. Abstract types are defined by abstract classes and do not directly correspond to objects.

We say that Sather is *strongly typed* because each variable has a type which specifies exactly which objects it can hold. We say that it is *statically type-safe* because it is impossible for a program which compiles to assign an object of an incorrect type to a variable. The Sather type correctness checking is purely local and does not require a system-wide analysis. It is done by checking calls against the declared signatures in the interface of the type to which the call is applied. Statically-checked strong typing is fundamental to achieving both the performance and the safety goals of Sather.

Type safety is ensured because of a *conformance* requirement on the interfaces of types[3]. If the type $\mathcal{A}$ conforms to the type $\mathcal{B}$, then the interface of $\mathcal{A}$ is required to conform to the interface of $\mathcal{B}$. This means that for each signature in $\mathcal{B}$'s interface there is a conforming signature in $\mathcal{A}$'s interface.

Object-oriented dispatch means that the particular implementation for a routine call is made according to the type of the object the call is made on. This object may be thought of as the first argument of the routine. Within the routine, it is referred to as `self`. The type of `self`, denoted as `SAME`, is the type defined by the class that implements the operation.

Under the subtype relation, the `self` parameter is *covariantly* typed. Because of the dispatching, this is typesafe[3]. All other arguments are *contravariantly* typed and the return value is *covariantly* typed. Together, these conformance requirements ensure that if a call is type correct on the declared type of a variable, then it will be type correct when made on all possible objects that may be held by that variable.

Sather allows for *multiple subtyping*: A type can be subtype of more than one type. This is very important for using software types to model types in the world. Real-world types are often subtypes of more than one type. In a system which only supports single subtyping, one is often forced to introduce spurious subtype relations which can destroy the conceptual integrity of a design.

A new feature introduced by Sather is the possibility for a new class to declare itself as a *supertype* of an existing class. Using this facility, it is possible to interpose a new type between two existing types in a hierarchy. This solves an old dilemma of class hierarchy design. On the one hand, for future flexibility one often wanted to introduce many incrementally different types. On the other hand, huge type hierarchies with many similar classes are hard to understand and use. With the ability to insert new types into a hierarchy, intermediate classes can be introduced only when needed.

2.2 Code Inheritance, Subclassing and Multiple Subclassing

Although often confused or combined with subtyping, an entirely different aspect of object-oriented programming is code reuse by means of *code inheritance*, also called *subclassing*. A class `A` is called a subclass of a class `B` if `A`'s implementation is based in part on `B`'s implementation. Code reuse in this sense differs from the use of traditional library routines in two important ways. First, the inherited code

[3] The language Cecil[4] uses multi-methods to allow multiple covariantly typed parameters in a type-safe way. Some disadvantages of multi-methods are discussed in section 2.7.

has direct access to the internal representation of the reusing class. Second, the inherited code may make calls on `self`. Such calls may call other inherited operations or operations explicitly defined in the new class. This intricate tangling of new and old code is powerful but complexity-prone [16].

As with subtyping, Sather allows *multiple subclassing*: A class can be subclass of multiple classes, i.e. it can reuse portions of the implementations of multiple classes. Multiple subclassing introduces many complications that require careful attention. Most languages combine multiple subtyping with multiple subclassing into *multiple inheritance*. The complexity introduced by multiple subclassing has given rise to widespread ambivalent feelings about multiple inheritance. A particularly tricky situation arises when the same code is inherited by a class along multiple paths. The resulting conflicts and Sather's conflict resolution mechanisms are described below in Section 3.

One could imagine introducing a construct for code inheritance which is analogous to the supertyping construct described above (cf. Section 2.1). This would be a form of "code injection" in which classes could add implementation to other classes. This possibility was rejected in the Sather design because it gives rise to many ambiguities and errors which would be hard to find. One would no longer be able to determine the source code of a class by merely looking at the class text and those classes reachable from references in it. When using classes from another system, it would not be clear which source files contributed code to the desired classes. Also, because of the separation between subclassing from subtyping, code can be inherited in the opposite direction from subtyping if desired.

2.3 Separating Subtyping and Subclassing

Traditionally, object-oriented languages are either untyped – e.g. Smalltalk [10] or Self [27] – or tightly bind classes and types – e.g. C++ [8], Eiffel [17] Modula-3 [21], or Oberon-2 [19]. (In contrast to Oberon-2, Oberon [23] keeps the dispatching of implementation variants separate from subtyping issues, essentially by not providing methods at all. Instead, Oberon relies entirely on procedure variables to implement late binding. Nevertheless, Oberon still does not completely separate subtyping from subclassing, cf. Section 2.6.)

The decision to have static type safety caused us to reject the untyped variants. Given that there will be types, one must decide how tight a binding there should be between subtyping and subclassing. The typed object-oriented languages mentioned above bind these notions closely together. Not separating these concepts properly leads to several problems, however.

One approach requires that every subclass relationship obeys the rules of type-safe subtyping. This leads to contravariant typing of routine arguments. It has been argued that this eliminates several important opportunities for code reuse [18, 14].

Another approach introduces subclasses which are subtypes by declaration but not in terms of the interface which is supported. This approach is adopted as a compromise in many languages, including the original version of Sather and Eiffel [17]. This violates the requirement of local type checkability. In the original Eiffel design this was a safety loop-hole [5]. The latest version of Eiffel requires "system-level type checking", which gives up on local type checkability and sometimes rejects dynamically type-safe programs.

Because of these problems, [6] suggested that subtyping should be clearly separated from subclassing. Emerald [11] is one of the few languages that actually implemented this separation. In Emerald, however, the result is a significant burden on the programmer. Often, subtyping and subclassing do go along in parallel, and Emerald requires separate specification even for this common case.

Later language designs, such as Sather 1.0 and Cecil [4], attempt to provide more convenient ways to support the common case. Since Cecil is based on prototype objects, quite similar to Self, its code inheritance is not based on classes. Still, Cecil's counterpart to subclassing has the default behavior of also introducing a subtype. This behavior can be explicitly prevented, however, and it is even possible to have code inheritance and subtyping go in opposite directions. Sather follows a similar path of optimizing the common case. However, instead of introducing defaults, Sather introduces special kinds of classes and an explicit means to implement subclassing and subtyping graphs over these classes.

2.4 Sather Types, Classes, and Variables

As described above, Sather distinguishes between *abstract* and *concrete* types (the names of abstract types are distinguished by a leading "$" to help distinguish them). Abstract classes can have descendants in the type graph, but cannot be instantiated as objects. Concrete classes are always leaf-nodes in the subtype graph, but can be instantiated. This approach is similar to the type system formally defined in [7]. Abstract classes may provide partial implementations to be inherited by subclasses, while concrete classes are required to fully implement their type. Sather code inheritance is explained in Section 3.

All Sather variables are statically typed. If a variable is declared as a concrete type, then only objects of exactly that type can be held by it. As a result, all calls on such variables are monomorphisms, i.e. the actual implementation invoked is statically determined. This is an important source of efficiency for Sather programs. If a variable is declared by an abstract type, then it can hold objects belonging to any of the subtypes of the declared type. Calls made on such "abstract variables" are polymorphisms. This means that the actual implementation invoked is determined at run-time according to the type of the object bound to the variable at the time of the call.

2.5 Examples of Separate Subtyping and Subclassing

Multiple subtyping is important in situations where there is not an obvious hierarchy of object properties. In the Sather library some container classes are internally based on hash tables, others are not. Not every object defines a corresponding hash function, however. We make objects which do provide a hash function be descendants of the abstract class `$HASHABLE` whose interface defines the single routine "`hash`":

```
abstract class $HASHABLE is
   hash:INT;
end
```

Note that `$HASHABLE` doesn't provide an implementation of `hash`, because there is no generic hash function that works for all types. Abstract classes without implementation information, such as `$HASHABLE`, only serve for subtyping purposes. The implementation of the required features is left to the descendants. Figure 1 shows a typical inheritance graph for defining an element class to be used as a type parameter in a parameterized, hash table-based set class (cf. Section 2.8. In Figure 1, as well as in the other inheritance graphs, solid and dashed arcs are used to represent subtype and subclass relationships. Class names set in a plain typeface denote abstract types and those in a bold typeface denote concrete types. (cf. Section 2.3).

Figure 1: Typical inheritance graph for `$HASHABLE`

Multiple subclassing is much less common in Sather programs than multiple subtyping. Nevertheless, there are situations where application programmers prefer to use multiple subclassing. It is used in the *mixin* programming style used extensively in CLOS[2]. ICSIM, the ICSI neural network simulator, uses this style to let the user configure the properties of neuron sites. Sites are subsets of a neuron's connections with identical properties. Sites have connection-oriented properties represented by `$PORT` descendants and computation-oriented properties represented by `$COMPUTATION` descendants. `$SITE` is a subtype of both `$PORT` and `$COMPUTATION`.

Typically, ICSIM users do not program their own sites, but instead choose them from built-in classes that provide building blocks for connection and computation code. All types used in ICSIM are also subtypes of `$ANY_ICSIM`. These relationships are shown in the inheritance graph in Figure 2. It is interesting to note that the MY_SITE class uses multiple subclassing but single subtyping (the opposite of the usual case).

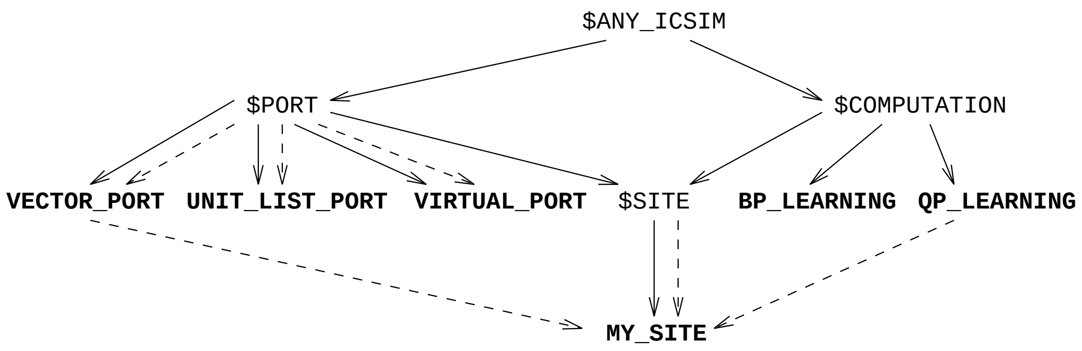

Figure 2: Multiple Subclassing for Programming by Configuration

2.6 Features of a Class

The features of a Sather class are either *attributes*, *routines*, or *iters*[4]. Attributes are the analog of record fields in Pascal-like languages. Routines are the equivalent of "methods" in some other object-oriented languages. In particular, routines differ from Pascal-like procedures by having an additional implicit parameter bound to the object that the routine is called on.

Whether a particular operation is implemented as an attribute or as a routine is not visible from the interface of a type. One concrete descendant of an abstract class may define an attribute while another may just provide accessor and modifier routines. This is a departure from the traditional coincidence of type and structure definition in Pascal-like languages, including Oberon. It is used in some earlier object-oriented languages, however, such as Self. One might argue that the same effect can be achieved by not introducing public attributes, as, for example, can be done in Oberon by not exporting record fields. The price to pay for doing so is the loss of the intuitive and lightweight attribute access notation "`x.a`" in clients of the class.

Attributes may be declared to be *shared* among all instances of a type. Such shared attributes serve the function of global variables (and the rather difficult to use "once functions" of Eiffel). Shared attributes may specify an initialization expression that is evaluable at compile-time. Similar attributes may be declared *constant*, in which case the binding established by the initializer is permanent.

The features of a Sather class may be declared *private*, allowing only the routines within the class to access them, and only relative to `self`. For attributes, it is possible to declare the accessor and modifier routines individually as private or public. This allows attributes to be read-write, read-only, or invisible from within code external to the class.

2.7 Object-Oriented Dispatch

Variables declared by an abstract type can hold objects of any descendant type. Routine calls made on such variables dispatch on the runtime type of the object to determine the code to execute. This lookup adds a small amount of extra overhead to such calls. By declaring a variable with an abstract or a concrete type, the programmer may decide to pay the price for routine dispatch or to restrict the generality of the code by precisely specifying the object type that the variable can hold.

Some languages support "multi-methods" which can dispatch on all the arguments of a call. Sather does not adopt this approach for both semantic and performance reasons. In Sather routines are grouped into classes according to the type of "`self`". This provides a natural organization principle and is responsible for the encapsulation of functionality into types. The interface of a type encapsulates the

[4] We do not describe iters here because it would take us too far afield and they have been described elsewhere [20].

abstraction defined by that type. With multi-methods code does not naturally belong to a particular type. Sather deals with multi-method situations by using "`typecase`" statements. These appear in the body of a routine which dispatches on the first argument type and may explicitly dispatch on the second argument type. Unlike a simple "`case`" statement applied to the type, a "`typecase`" statement can branch on abstract types. This means they can be used in the same situations that multi-methods would be helpful. This approach also makes the performance consequences of a multi-method organization explicit rather than hiding it behind a complex language construct.

Sather routines can also be called directly. A direct call is equivalent to dispatching the routine call on an unbound variable of concrete type (`self = void`). Direct-called routines are Sather's version of plain procedures in Pascal, class methods in Smalltalk, and static member functions in C++.

2.8 Parameterized Classes

Sather allows the definition of a family of classes parameterized by types. This is a similar mechanism to the generic packages of Ada [28] and templates in the newer versions of C++. Sather type parameters have associated type constraints. The values specified for the type parameters are required to be subtypes of these constraint types. The supertyping feature introduced in Section 2.1 is quite useful for defining such constraints. A constraint type representing an arbitrary union of types can be introduced by forming an appropriate supertype.

A second form of genericity in Sather is related to the typing of arguments and return values in inherited code. Sather allows such types to be declared as `SAME`, similar to Eiffel's `like-current`. If a class `A` inherits code which refers to the type `SAME`, it behaves as if the type were replaced by `A`. For a subclass to be also a subtype, however, this replacement has to follow the subtyping rules stated in Section 2.1. This is very different from Eiffel's `like-current`, where a subclass formed in this way is automatically considered a subtype, even though it might well have introduced a conformance conflict.

2.9 Reference and Value Classes

Sather distinguishes between *reference*, *value*, and *bound* objects. Most user-defined objects are reference objects. These are passed by reference as routine arguments and may be aliased. The fundamental types representing boolean values, integers, characters, floating point values, etc. are called *value* objects. These are always passed by value and it is not possible to alias them (i.e. to reference the same object under two names). More pure object-oriented languages such as Smalltalk and Self try to unify these notions.

Languages that only operate over values are typically called *functional languages*, and operations defined only over value types are side-effect free and therefore *referentially transparent*. On the other hand, reference objects are best used to model entities that have an identity plus a current state. The idea of an object identity bound to a modifiable state introduces referential opaqueness and allows for side-effects.

Sather distinguishes between these at the level of types. Instances of value types have value semantics: Once created they never change, and there is no such thing as a "reference" to a value object. Reference objects have an identity and the state of a reference object can be modified by writing to its attributes. A variable of abstract type can be used to store either value or reference objects.

The special properties of value objects make them especially amenable to compiler optimization techniques. Most important, a value object can be copied freely without the possibility of aliasing conflicts. Logically, when value objects are passed as arguments, their value is first copied and then the operation is invoked on the copy (*call by value* semantics). Of course, the compiler is free to eliminate this copying whenever it can deduce that the invoked operation cannot modify the object.

The introduction of separate value and reference classes imposes certain restrictions on subclassing: An abstract class can only be a subclass of other abstract classes; a value class can only be a subclass of abstract classes and other value classes; a reference class can only be a subclass of abstract classes and other reference classes.

2.10 Bound Routines

A controversial feature of non-functional programming languages are closures or higher-order functions. While expressive and powerful, certain formulations are difficult to implement efficiently. Hence, many non-functional programming languages provide more lightweight but much less powerful facilities.

Pascal [12] introduced procedure parameters, but no procedure variables. This allowed implementations to strictly adhere to a stack discipline, but prevented the use of procedures as first-class values in data structures. In Modula-2 [29] this was changed to allow for procedure variables, but the restriction was added that only global procedures can be assigned. C [13] has function pointers with a similar semantics. While this allows first-class procedure values, it restricts such procedures to operate on the global state only, while in Pascal it was possible to pass a nested procedure that in turn could operate on the current bindings of local variables of the passing procedure.

Sather, as in C, has no nested routines, hence the C / Modula-2 solution would work without any constraints. However, the constraint that routines bound to a variable can only operate on the global state is much weaker than many applications need. For example, to implement a routine which produces the complement of a boolean argument routine or the composition of two argument routines there must be internal state associated with the routine.

The Sather solution is to introduce *bound routines*[5] to express higher-order functions and closure-like constructs. The key idea is that the parameters of a routine, including the implicit `self`, can be bound to objects. The resulting bound routine can then be assigned to a routine variable of the appropriate type. For example, it is possible to take a routine with two integer parameters, bind one of these to an integer value, and then assign the resulting bound routine to a variable that asks for a routine with a single integer parameter. Bound types describe the resulting signature of a bound routine. Conformance is defined as contravariant conformance of the type signature.

3 Code Inheritance

3.1 The Textual Inclusion Model for Code Inheritance

The semantics of *code inheritance* in Sather is defined by textual inclusion of the inherited code. So-called "`include`" clauses are used to incorporate source code from a specified class. The choice of the keyword "`include`" was made to indicate the textual semantics for the inheritance model. References to the type "`SAME`" in the inherited code represent the type of the inheriting class. Newly defined features in a class override inherited features with a conforming signature (as defined in Section 2.1). This approach differs from that used in Smalltalk and most other object-oriented languages, in which a call conceptually climbs up in the class hierarchy until a corresponding method is found. For most common cases, the two approaches produce identical results. In complex situations, however, the textual inclusion approach seems easier to understand and to reason about.

It is sometimes convenient for a new version of a feature to call the old version that it overrides. Smalltalk solves this problem by providing the "super"-call, which bypasses any matching implementations in the object's class and passes the call directly to the superclass. We found that in Sather, this approach would be confusing in certain circumstances. The problem arises when code which makes a super call is itself inherited. The ambiguity for programmers was whether the inherited "super" call refers to the "super" class of the original defining class or of the inheriting class.

To eliminate this problem, Sather replaces the "super"-call approach with a general "renaming" facility in the `include` clauses which define code inheritance. The `include` clause comes in two forms: one is used to include and possibly rename a single feature from another class and the other includes an entire class but may cause features to be undefined or renamed. Renaming is shallow, i.e. renaming affects only the definition of the specified feature but not calls on that feature[6]. Appendix A.10 includes the syntax of the construct. Figure 3 uses the example of extending a simple unit (neuron) in ICSIM to a unit with back-propagation learning to show how the "super"-call problem is solved in

[5] Sather also introduces *bound iters*.

[6] Thus, renaming or undefining a feature may break inherited code. If this is the case the compiler signals a "subclassing error" associated with the corresponding include clause.

Sather. The routine `accumulated_input` inherited from `SIMPLE_UNIT` is renamed as the private routine: "`SIMPLE_UNIT_accumulated_input`" in `SIMPLE_BP_UNIT`.

```
class SIMPLE_UNIT is
  ...
  accumulated_input: REAL is
     -- Compute the dot product of input values * weights
     input_port.get_outputs_into_vec(input_values);
     res := input_values.dot_v(weights)
  end;
  ...
end;

class SIMPLE_BP_UNIT is
  include SIMPLE_UNIT
    accumulated_input -> private SIMPLE_UNIT_accumulated_input ... ;
  ...
  accumulated_input: REAL is
     -- Compute the dot product of weights * inputs + the bias value
     res := SIMPLE_UNIT_accumulated_input + bias.val
  end;
  ...
end
```

Figure 3: Using renaming instead of a "super"-call.

One may argue that the renaming solution for "super"-calls unnecessarily clutters the name space. Our experience shows that we use this style of programming infrequently, and if we need it we make the renamed version of the old routine `private` in order not to affect the external interface of a class. Because every routine has a specified name, the approach eliminates any ambiguity in the interpretation of code. As shown in the next section, the renaming approach is also more general than the "super"-call approach.

3.2 Multiple Subclassing and Conflict Resolution

Sather supports *multiple subclassing* (multiple code inheritance) by allowing multiple `include` clauses per class. Since more than one of the superclasses may provide a feature with the same signature, multiple subclassing leads to *inheritance conflicts*. Two routines or iters are said to *conflict* if they have the same name, the same number and types of arguments, and both either have or do not have a return value. Reference [9] describes four ways to cope with inheritance conflicts:

1. *Disallow conflicts*: Signal an error in the case of a conflict.
2. *Resolve conflicts by explicit selection*: Require the user to make a selection in case of a conflict. This is Sather's approach, as described below.
3. *Form disjoint union of features*: Create a separate feature for each conflicting feature. This is the approach of C++ where feature names of sub- and superclasses are in different scopes. The user selects between conflicting features using the scope resolution operator "`::`".
4. *Form composite union of features*: Create one single feature for each conflicting feature by algorithmically resolving the conflict. CLOS [1] follows this approach by linearizing the class hierarchy.

1. to 3. are *explicit* conflict resolution methods, 4. is an *implicit* method. Cecil [4] takes an intermediate stance between 3. and 4. by imposing only a partial ordering on classes, and requiring any remaining

conflicts to be resolved explicitly by the programmer. We agree with [25] that CLOS-style linearization of the inheritance graphs may lead to unexpected method lookups, and result in faulty and hard to debug programs.

Sather, therefore, adopts an explicit conflict resolution scheme in which the programmer has to explicitly choose in case of conflicts. A class may not explicitly define two conflicting routines or iters. A class may not define a routine which conflicts with the reader or writer routine of any of its attributes (whether explicitly defined or included from other classes). If a routine or iter is explicitly defined in a class, it overrides all conflicting routines or iters from included classes. The reader and writer routines of a class's attributes also override any included routines and must not conflict with each other. If an included routine or iter is not overridden, then it must not conflict with another included routine or iter. Renaming or undefining in include clauses is used to resolve these conflicts.

Any language which supports code inheritance must deal with the problem of the same code inherited along two different paths. Some languages introduce complex mechanisms to deal with this case, but these tend to be confusing to programmers and rarely do exactly what is desired. Sather's solution is implied by the rules given above. Sather does not consider the origin of code and resolves inheritance solely based on the body of the class itself and the bodies of the classes it includes (*after* their own code inheritance has been resolved). This behaves like the non-virtual inheritance of C++ for diamond-shaped inheritance graphs, i.e. features from a common superclass are included along each edge. This sometimes necessitates explicitly choosing a single version of a routine inherited along multiple paths, but it eliminates complex rules which depend on the structure of the code inheritance graph.

Our experience with the Sather libraries is that we use multiple subclassing only rarely. We therefore felt that these special cases were too weak a justification to introduce a complex graph-based subclassing scheme or a strategy based on structural equality of feature definitions.

3.3 Separate Compilation

Sather has no explicit notion of structural units comprising multiple classes. The Sather programming environment is intended to manage and maintain the source code of multiple classes. In particular, when compiling a new class it is often required that the Sather compiler has access to the source code (or at least the type interface and dependency information) of all classes referred to by it.

For example, the compiler automatically inlines short routines to improve efficiency. There tend to be many short routines in object-oriented programming because a routine which is needed only for the purposes of an abstract interface often just calls another routine. In addition to eliminating an extra routine call, inlining allows much more optimization to be done within a routine with inlined code. On the one hand, compiler-controlled inlining requires that the code to be inlined is available to the compiler and to the compiler's analysis process, i.e. that the source is at hand. On the other hand, inlining introduces hidden dependencies between implementations.

For large systems, there are arguments for introducing another level of modularity. In some cases, one doesn't want to require that all source code be available or allow arbitrary dependencies between compiled units. Such large systems are usually composed of subsystems. For a limited subsystem the global analysis is acceptable. For a composed system, however, it should be possible to define the subsystems in a way that global analysis is not required.

For Sather, it is possible to form subsystems with strict boundaries in terms of compiler analysis. Such a subsystem must be limited by an interface presenting only types, i.e. empty abstract classes, to subtyping clients, and allowing for direct calls to routines (c.f. Section 2.7) defined by classes within the subsystem.

The most prominent mechanism that cannot be allowed to cross subsystems is code inheritance, which of course is a direct consequence of specifying the semantics of code inheritance based upon the actually inherited source text. Also to be excluded from a subsystem's interface are parameterized classes: The current Sather compiler cannot completely check a parameterized class before its parameters actually get specified. This defect in the checkability of Sather's parameterized classes is unfortunate and an issue of ongoing research. However, this problem is not specific to Sather, the same holds for C++, where such errors might be detected as late as at link-time(!), Eiffel, and Ada. Possible solutions tend to either restrict the usefulness of parameterized classes, or to introduce a complicated apparatus to specify sufficiently strong bounds on the parameters.

Explicit support for expressing subsystem boundaries, such as modules [26], might be a useful extension to Sather. In particular, a module construct would help to package helper classes, to explicitly treat subsystem invariants, to reduce the probability of conflicts in the global class name space, and allow limitations to be placed and how much of a source needs to be revealed for purposes of compilation. The best form for such a construct is not yet clear, however, and so the current version of Sather will address these issues at the level of the development environment rather than in the language.

4 Foundations

The type system described so far needs to be grounded in explicit built-in classes. A class can do no more than define attributes of types introduced by itself or other classes, or define routines operating over its own or shared attributes, or invoking other operations. What is missing are the foundational entities to start with. Such foundation entities are present in all languages[7] in the form of built-in types or operations with predefined semantics. In Sather, certain predefined classes serve this purpose.

A language that claims to be "general-purpose" also has to be able to express interfaces to the outside world. For example, a Sather program should be able to call non-Sather libraries, including functions of the underlying operating system and graphical user interface. It is not reasonable to expect that a fixed set of built-in classes will ever suffice to serve this purpose in full generality. For these purposes Sather has *external classes*. Predefined and external classes are described in the next two subsections.

4.1 Built-in Classes

Most classes are defined by explicit code in a Sather program, but there are several classes which are automatically constructed by the compiler. These classes have certain built-in features that may be defined in an implementation dependent way. In each case, the choices made by the implementation are described by constants which may be accessed by a program. This section provides a short description of some of the most important built-in classes. The complete and detailed semantics and precise interface is specified in the Sather class library documentation.

- `$OB` is automatically an ancestor of every class. Variables declared by this type may hold any object.
- `BOOL` defines value objects which represent boolean values.
- `CHAR` defines value objects which represent characters.
- `STR` defines reference objects which represent strings.
- `INT` defines value objects which represent machine-dependent integers. The size is implementation dependent. Classes representing fixed-sized integers with a different number bits may be defined by inheriting from `INT` and redefining the constant "`bsize`". All the routines work with an arbitrary "`bsize`".
- `INTINF` defines reference objects which represent infinite precision integers. They support arithmetic operations but do not support bit operations.
- `FLT`, `FLTD`, `FLTE`, and `FLTDE` define value objects which represent floating point values according to the single, double, extended, and double extended representations defined by the IEEE-754-1985 standard.
- `ARR{T}` is a reference class defining dynamically-sized arrays of elements of type `T`. Classes which inherit from this are called *array classes*. They allocate space for the array and the attribute `asize:INT` whose value is the number of elements in the array.
- `TYPE` defines the value objects returned by the `type` routine.

[7] In theory, the λ-calculus, e.g. with syntax $E ::= x \mid EE \mid \lambda x.E$, is sufficient. Such languages tend to have efficiency problems, though!

4.2 Interfacing to External Code

Sather provides a few special built-in classes to interface to external code, as listed below. Additionally, Sather's external classes can be used to interface with code from other languages. External classes are not classes in the traditional sense. They can neither be instantiated, nor can they be in a subclass or subtype relationship with any other class. It is merely for the sake of uniformity of the language that external routine interfaces are grouped into external "classes".

Each external class is typically associated with an object file compiled from a language like C or Fortran. External classes may only contain routines with distinct names (overloading is not allowed in external classes). The external object file must provide a conforming function definition with the same name as each routine which doesn't have an implementation in the external class. Sather code may call these external routines using a class call expression of the form `EXT_CLASS::ext_rout(5)`. Similarly, the external code may call one of the non-abstract Sather routines[8] defined in the class by using a name consisting of the class name, an underscore, and the routine name (eg. `EXT_CLASS_sather_rout`).

- `BITS` may be inherited by value classes which represent a single field of data. The descendant may define the two constants `bsize:INT` and `balign:INT` to specify the size in bits of the object and its alignment requirements.
- `$EXTOB` is used to refer to "foreign pointers". These might be used, for example, to hold references to C structures. Such pointers are never followed by Sather and are treated essentially as integers which disallow arithmetic operations. They may be passed to external routines.

5 Conclusions

The design of Sather 1.0 involved trading off an interesting set of constraints regarding efficiency, clarity, reusability and safety. We have described several important aspects of the type and class system and compared them with the solutions chosen by other object-oriented languages. These give rise to a language with a unique combination of conceptual clarity, safety and support for high performance.

Acknowledgements

Many people were involved in the Sather 1.0 design discussions. Jerry Feldman, Ben Gomes, Ari Huttunen, Chu-Cheow Lim, Heinz Schmidt, and David Stoutamire made suggestions which were especially relevant to the topics discussed in this paper.

A Syntax of the Sather Class and Type System

The following sections give examples of actual Sather code fragments together with the corresponding grammar rules. The grammar rules are presented in a variant of Backus-Naur form. Non-terminal symbols are represented by strings of letters and underscores in italic typeface and begin with a letter. The nonterminal symbol on the lefthand side of a grammar rule is followed by an arrow " ⇒ " and right-hand side of the rule. The terminal symbols consist of Sather keywords and special symbols and are typeset in the `typewriter` font. Italic parentheses "*(...)*" are used for grouping, italic square brackets "*[...]*" enclose optional clauses, vertical bars "... | ... " separate alternatives, italic asterisks "... *" follow clauses which may appear zero or more times, and italic plus signs "... *+*" follow clauses which may appear one or more times.

A.1 Class definition lists

`class A is ... end; class B is ... end`

class_def_list ⇒ *[class_def]* | *class_def_list* `;` *[class_def]*

[8] The calling conventions and the layout of objects are described in the implementation manual of individual versions.

A.2 Class definitions

```
class A{S,T:=INT,U<B} is ... end
value class B < $C,$D is ... end
abstract class $E > G,H is ... end
```

class_def ⇒ *[*`value` | `abstract` | `external`*]* `class` *class_name*
  [`{` *param_dec (*`,` *param_dec)**`}`*]* *class_inheritance* `is` *class_elt_list* `end`

param_dec ⇒ *ident [*`<` *type_spec] [*`:=` *type_spec]*

class_inheritance ⇒ *[*`<` *type_spec (*`,` *type_spec)*]*
  [`>` *type_spec (*`,` *type_spec)*]*

A.3 Type specifiers

```
A{B,C{$D}}
ROUT{A,B,C}:D
ITER{A,B!,C}
```

type_spec ⇒ *[class_name] [*`{` *type_spec_list* `}`*]* |
  `ROUT` *[*`{` *type_spec_list* `}`*] [*`:` *type_spec]* |
  `ITER` *[*`{` *type_spec [*`!`*] (*`,` *type_spec [*`!`*])**`}`*] [*`:` *type_spec]*

type_spec_list ⇒ *type_spec (*`,` *type_spec)**

A.4 Features

class_elt_list ⇒ *[class_elt]* | *class_elt_list* `;` *[class_elt]*

class_elt ⇒ *const_def* | *shared_def* | *attr_def* | *rout_def* | *iter_def* | *include_clause*

A.5 Constant attribute definitions

```
const r:FLT:=45.6
private const a,b,c
```

const_def ⇒ *[*`private`*]* `const` *ident (*`:` *type_spec* `:=` *expr* | *[*`:=` *expr] [*`,`*ident_list])*

ident_list ⇒ *ident (*`,` *ident)**

A.6 Shared attribute definitions

```
private shared i,j:INT
shared s:STR:="name"
readonly shared c:CHAR:='x'
```

shared_def ⇒ *[*`private` | `readonly`*]* `shared`
  (ident `:` *type_spec* `:=` *expr* | *ident_list* `:` *type_spec)*

A.7 Object attribute definitions

```
attr a,b,c:INT
private attr c:CHAR:='a'
readonly attr s:STR:="a string"
```

attr_def ⇒ *[*`private` | `readonly`*]* `attr`
  (ident `:` *type_spec* `:=` *expr* | *ident_list* `:` *type_spec)*

A.8 Routine definitions

```
a(FLT):FLT pre arg>1.2 post res<4.3 is ... end
b is ... end
private d:INT is ... end
c(s1,s2,s3:STR)
```

rout_def ⇒ *[*`private`*] ident [*`(` *arg_dec (*`,` *arg_dec)*)`)`*] [*`:` *type_spec] [*`pre` *expr] [*`post` *expr] [*`is` *stmt_list* `end`*]*

arg_dec ⇒ *[ident_list* `:`*] type_spec*

A.9 Iter definitions

```
elts!(i:INT, x:FLT!):T is ... end
```

iter_def ⇒ *[*`private`*] iter_name [*`(` *iter_arg_dec (*`,` *iter_arg_dec)**`)`*] [*`:` *type_spec] [*`pre` *expr] [*`post` *expr] [*`is`*stmt_list* `end`*]*

iter_name ⇒ *ident*`!`

iter_arg_dec ⇒ *[ident_list* `:`*] type_spec [*`!`*]*

A.10 `include` clauses

```
include A a:INT->b, c(INT)->, d:FLT->private d;
private include D e:STR->readonly f;
include A::a(INT)->b;
```

include_clause ⇒ `include` *type_spec* `::` *elt_mod* | *[*`private`*]* `include` *type_spec [elt_mod (*`,` *elt_mod)*]*

elt_mod ⇒ *ident [*`(` *type_spec_list* `)`*] [*`:` *type_spec]* `->` *[[*`private` | `readonly`*] ident]*

References

[1] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. Keene, G. Kiczales, and D. A. Moon. The common lisp object system specification. Technical Report 88-002R, X3J13, June 1988. Also in special issue of SIGPLAN Notices 23 (Sep. 1988) and Lisp and Symbolic Computation (Jan. 1989).

[2] Gilad Bracha and William R. Cook. Mixin-based inheritance. In *Proceedings of the Conference on Object-Oriented Programming, Systems, and Applications and European Conferance on Object-Oriented Programming (OOPSLA/ECOOP'90)*, Ottawa, Canada, October 1990. Also in SIGPLAN Notices, 25:10, Oct. 1990.

[3] Luca Cardelli. Typeful programming. Technical report, DEC Systems Research Center, Palo Alto, CA, May 1989.

[4] Craig Chambers. The Cecil language - specification and rationale. Technical Report 93-03-05, Department of Computer Science, University of Washington, Seattle, WA, March 1993.

[5] William R. Cook. A proposal for making eiffel type safe. In *Proceedings of the Third European Conference on Object-Oriented Programming (ECOOP'89)*, pages 57–70, Nottingham, England, 1989. Cambridge University Press.

[6] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL'90)*, pages 125–135. ACM Press. Addison-Wesley, 1990.

[7] Mahesh Dodani and Chung-Sin Tsai. ACTS: A type system for object-oriented programming based on abstract and concrete classes. In *Proceedings of the Sixth European Conference on Object-Oriented Programming (ECOOP'92)*, pages 309–328, Utrecht, Netherlands, 1992.

[8] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[9] Richard P. Gabriel, Jon L White, and Daniel G. Bobrow. CLOS: Integrating object-oriented and functional programming. *Communications of the ACM*, 34(9):29–38, September 1991.

[10] Adele Goldberg and David Robson. *Smalltalk-80, The Language and its Implementation.* Addison-Wesley, 1985.

[11] Norman Hutchinson. *Emerald: An Object-Oriented Language for Distributed Programming.* PhD thesis, Department of Computer Science and Engineering, University of Washington, Seattle, WA, January 1987.

[12] Kathleen Jensen and Niklaus Wirth. *PASCAL: User Manual and Report.* Springer-Verlag, 2d ed. corr. print edition, 1978.

[13] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

[14] B.B. Kristensen, O.L. Madsen, B. Moeller-Pedersen, and Kristen Nygaard. The BETA programming language. In B.D. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming.* MIT Press, 1987.

[15] Chu-Cheow Lim and Andreas Stolcke. Sather language design and performance evaluation. Technical Report TR-91-034, International Computer Science Institute, May 1991.

[16] Boris Magnusson. Code reuse considered harmful. *Journal of Object Oriented Programming*, 4(3), November 1991.

[17] Bertrand Meyer. *Eiffel - The Language.* Prentice-Hall, 1988.

[18] Bertrand Meyer. *Object-oriented Software Construction.* Prentice-Hall, 1988.

[19] Hanspeter Mössenböck and Niklaus Wirth. The programming language Oberon-2. *Structured Programming*, 12(4), 1991.

[20] Stephan Murer, Stephen Omohundro, and Clemens A. Szyperski. Sather iters: Object-oriented iteration abstraction. Technical Report TR-92-xxx, International Computer Science Institute, 1993.

[21] Greg Nelson, editor. *Systems Programming with Modula-3.* Prentice Hall, 1991.

[22] Stephen Omohundro. Sather provides nonproprietary access to object-oriented programming. *Computers in Physics*, 6(5):444–449, 1992.

[23] Martin Reiser and Niklaus Wirth. *Programming in Oberon. Steps Beyond Pascal and Modula.* Addison-Wesley, 1992.

[24] Heinz W. Schmidt and Benedict Gomes. ICSIM: An object-oriented connectionist simulator. Technical Report TR-91-048, International Computer Science Institute, November 1991.

[25] Alan Snyder. Encapsulation and Inheritance in object-oriented programming languages. In *Proceedings of the First ACM Conference on Object-Oriented Programming, Systems, and Applications (OOPSLA'86)*, pages 38–45, Portland, OR, November 1986. Also in SIGPLAN Notices, 21:11, Nov. 1986.

[26] Clemens A. Szyperski. Import is not Inheritance – why we need both: Modules and Classes. In *Proceedings of the Sixth European Conference on Object-Oriented Programming (ECOOP'92)*, Utrecht, The Netherlands, June 1992.

[27] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Proceedings of the Second ACM Conference on Object-Oriented Programming, Systems, and Applications (OOPSLA'87)*, Orlando, FL, October 1987. Also in SIGPLAN Notices, 22:12, Dec. 1987.

[28] U.S. Department of Defence. *Ada Reference Manual: Proposed Standard Document*, July 1980.

[29] Niklaus Wirth. *Programming in Modula-2*. Springer-Verlag, 1982.