The Sather Language and Libraries

Stephen Omohundro *
ICSI
1947 Center St
Suite 600
Berkeley, CA 94704

Chu-Cheow Lim †
ICSI
1947 Center St
Suite 600
Berkeley, CA 94704

February 28, 1992

Abstract

Sather is an object-oriented language derived from Eiffel which is particularly well suited for the needs of scientific research groups. It is designed to be very efficient and simple while supporting strong typing, garbage collection, object-oriented dispatch, multiple inheritance, parameterized types, and a clean syntax. It compiles into portable C code and easily links with existing C code. The compiler, debugger and several hundred library classes are freely available by anonymous FTP. This paper describes aspects of the language design, implementation and libraries.

Category: Research Paper.

1 Introduction

Sather is a new object-oriented language derived from Eiffel which is designed to be very efficient and simple while supporting strong typing, garbage collection, object-oriented dispatch, multiple inheritance, parameterized types, and a clean syntax. Sather was initially developed to meet the needs of research projects at the International Computer Science Institute (ICSI) which required a simple, efficient, non-proprietary, object-oriented language. ICSI is involved in several areas which require the construction of complex software for computationally intensive tasks. Examples include a general purpose connectionist simulator, a high-level vision system based on complex geometric data structures, the support software for a high-speed parallel computing engine for speech recognition, and CAD tools for integrated circuit design.

We investigated several existing languages including C++ [13], Objective C [2], Eiffel [8] [9], Self [14], Smalltalk [6], and CLOS [5]. Only C++ was efficient enough for our needs but its overall complexity and lack of garbage collection, object type tags (needed for persistency and object distribution), and parameterized types led us to begin our work with Eiffel. Our experience with

*Email: `om@icsi.berkeley.edu`. Phone: 510-642-4274
†Computer Science Division, U.C. Berkeley. Email: `clim@icsi.berkeley.edu`. Phone: 510-642-4274

Eiffel allowed us to identify the features which were essential for our purposes. We required a non-proprietary compiler, however, to serve as a base for developing a parallel object-oriented language to run on new experimental parallel hardware.

Sather was developed to incorporate the features of Eiffel which were essential to us as well as others aimed at enhancing efficiency and simplicity. The initial Sather compiler was written in Sather over the summer of 1990. The compiler generates portable C code and easily links with existing C code. In June 1991, ICSI made the language publicly available by anonymous FTP over the internet (from "ftp.icsi.berkeley.edu" in the U.S. and "gmdzi.gmd.de" in Europe). The release includes documentation, a compiler, a symbolic debugger, a GNU emacs programming environment, and several hundred library classes. Within a few weeks of the release, several hundred research groups from around the world had obtained copies. Since that time new class development has become an international cooperative effort.

This paper describes our rationale for some of the decisions made in the language design, some of the implementation techniques we used, and some of the concepts we found useful in the construction of library classes.

2 The Language

This section can only describe a few of the features of Sather. The complete language specification is included in the software distribution [10]. Sather's overall structure is derived from Eiffel but it differs in many details. In general, decisions were made in the direction of conceptual simplicity and high computational efficiency. Sather does not have Eiffel's emphasis on "programming by contract" and has a much less complex approach to inheritance. In contrast to most object-oriented languages, Eiffel and Sather share the use of strong typing. Perhaps the most important distinction between Sather and Eiffel is that the Sather type system allows the programmer to explicitly distinguish between declarations that cause dispatch and declarations that are resolved by the compiler. We discuss some of the consequences of this decision below. A more detailed comparison of Sather, Eiffel and CLOS appears in [11].

The language is intentionally quite small so that it is easy to learn and to remember. The entire Sather YACC grammar fits on a single page. Part of the philosophy is that the language should provide the basic construction mechanisms but most of the complexity should reside in an extensive library of classes. This approach allows one to use only what one needs and to easily replace or modify functionality without altering the basic language. Functionality which is sometimes included

directly in a language (e.g. random number generation, mathematical special functions, and hash tables) are provided as well-encapsulated library classes in Sather.

2.1 Code Reuse

The primary benefits we wanted to obtain from object-oriented programming were code reuse and cleanliness of organization due to encapsulation. It is particularly true in a research environment that projects tend to share many subcomponents. As research progresses, projects may take unexpected turns which require building new software which is often quite similar to existing software. In addition to the benefit of not having to rewrite existing code, reuse leads to great improvements in code reliability and testing. A piece of code which is used in several projects tends to be better written and better tested than code which is used only once.

There are several kinds of code reuse possible in object-oriented programming. From our perspective, the most important is the ability for old code to call new code. The libraries of traditional languages (such as C or Fortran) support a form of reuse in that newly written code can easily make calls to old library code. It is not easy, however, for old libraries to make calls to newly written code during their operation. Sather supports two basic mechanisms for old code to call new code: object-oriented dispatch and parameterized classes.

Object-oriented dispatch is a runtime mechanism that chooses a routine to be called at a certain point based on the runtime type of an object. If the object belongs to a newly written class, the routine which is called may not even have existed when the calling routine was written. Because it is a runtime mechanism, there is a small performance overhead to perform the dispatch.

Parameterized classes are a compile-time mechanism. A class is written with certain types left as variables. When a parameterized class is used, these type variables are instantiated and the compiler generates appropriate calls. The type parameter in an existing class may be instantiated with a new type and the compiler will generate calls from existing code to new code.

2.2 Inheritance

Sather also uses multiple inheritance to allow new classes to be defined which reuse routine and attribute definitions from existing classes. This kind of reuse is often convenient and can be important in structuring systems. We have found, however, that there is often a strong tendency for programmers to identify reuse with inheritance and to overuse it. Inheritance is in some ways directly opposed to encapsulation. It tends to distribute code over many classes rather than cleanly

encapsulating it in one place. There seems to be a tendency to build systems with extremely complex interdependencies between classes. Such systems are very hard to modify and are difficult for others to understand. Although there is positive benefit for code in an ancestor to be used in many descendents, it also forces the additional constraint that changes to the ancestor code not break the descendants.

The Sather language, therefore, has a very simple and easy to understand approach to inheritance and does not provide many of the complex features in other languages. In Sather, a class may specify that it inherits from a list of other classes. The semantic effect of such a specification is exactly equivalent to textually copying the bodies of the inherited classes into the descendent class. Later feature definitions override earlier ones with the same name. A feature may be declared `UNDEFINE` to eliminate it. This approach has the virtue that the program text clearly shows what is defined in a class and what its definition is. More complex inheritance features of other languages such as: "before" and "after" methods, "whoppers" and "wrappers", repeated inheritance of a feature (allowing multiple copies of an attribute inherited along more than one path in the inheritance DAG), and renaming features along certain paths are not supported. In the Sather style of programming we have not felt the lack of these features.

2.3 Strong Typing

The Sather type system is stronger than most other object-oriented languages. In most languages if a variable is declared to be of type `T` it can legally hold objects whose type is any descendent of `T` in the type hierarchy. This is often very useful in that it allows new subtypes to be defined and acted upon by existing code. It also introduces computational inefficiency and conceptual ambiguity. There are many situations in which the type of object that a variable can hold is precisely known by the programmer. Sather allows one to distinguish between variables which can only hold objects of a particular type and variables which can hold any descendent of a declared type. Object-oriented dispatching is only performed on the second kind of declaration. When the type is precisely specified, the compiler generates direct calls which are exactly as efficient as corresponding C code would be.

The Sather compiler is itself a large program written in Sather (see section 3.4) which uses a lot of dispatching. Each of the nodes in the Sather code trees uses dispatching on its children (e.g. each type of statement and each type of expression is represented by a separate object type). The performance consequences of dispatching were studied by comparing a version in which all references are dispatched to the standard version [7]. The use of explicit typing causes an 11.3% reduction in

execution time and approximately 947.0 % reduction in the number of dispatches.

The Sather type system is stronger than that of most languages because it allows more distinctions to be specified. It therefore has more of the benefits and pitfalls of strong typing. Strong typing offers not only computational efficiency, but also conceptual advantages. The compiler is able to perform stronger type checking and so catch errors that would not be caught with weaker typing. It also allows Sather classes to be structured in ways which more naturally reflect the represented concepts.

For example, a simple two-dimensional geometry system might have a class `POLYGON` with descendants `TRIANGLE` and `SQUARE`. It might be important for the `POLYGON` class to define a routine `add_vertex` which increases the number of sides by one. Such a routine is not appropriate for the descendant classes `TRIANGLE` and `SQUARE` because they have a fixed number of sides. In Sather, these classes can undefine this inherited routine. If a variable declared to be a `POLYGON` could hold a `TRIANGLE` at runtime, the compiler could not check the possible application of the illegal routine `add_vertex` to a `TRIANGLE` object. Because Sather allows one to declare variables which can only hold `POLYGON` objects, `add_vertex` may be applied to them with complete safety.

This distinction is also relevant to the basic types representing integers, characters, real numbers, etc. In many object-oriented languages objects of these type have tags or tag bits which specify their type. At runtime, extra tag checking code must run in addition to any operations performed on these objects. This is wonderful from the point of view of conceptual purity, but is simply not acceptable for the performance goals of Sather. In Sather, objects declared to be of these basic types are guaranteed to actually hold them. This allows the compiler to generate code with no tag checking. In addition to saving the cost of the check, it allows the wide variety of optimizations that modern compilers provide. Much scientific computing code intensively computes with such types and in Sather the result is as efficient as in C. Another basic structure for which Sather emphasizes efficiency is arrays. The implementation uses arrays in a fundamental way and avoids much of the indirection found in other designs (see section 3.3).

3 The Implementation

This section describes several aspects of the implementation of the compiler and runtime environment. [7] extensively studied the performance of the Sather compiler on both the MIPS and Sun Sparc architectures. Because the Sather compiler uses C code as an intermediate language, the quality of the executable code depends on the match of the C code templates used by the Sather compiler

to the optimizations employed by the C compiler. Compiled Sather code runs within 10% of the performance of hand-written C code on the MIPS machine and is essentially as fast as hand-written C code on the Sparc architectures. On a series of benchmark tests (including examples like towers of Hanoi, 8 queens, etc.) Sather performed slightly better than C++ and several times better than Eiffel.

3.1 Dispatching

Perhaps the central identifying feature of object-oriented programming is the runtime dispatch to an appropriate routine or attribute based on the type of an object. Because of Sather's strong type system, typically many fewer dispatches are necessary in a Sather program than in an equivalent program in other object-oriented languages. Nevertheless, the performance of dispatching is critically important. The implementation uses a combination of a fast hash table and local caching to perform the dispatch. The input to a dispatched call is the runtime type of the object and a compiler generated index which encodes the called feature. The hash table can quickly convert these to a function pointer or offset (depending on what kind of feature has been called).

It is quite common that dispatched calls are made repeatedly to the same object at the same point in the code. For this reason two static variables are used to cache the result of the hash table lookup and the object type from one call to the next. The generated code first compares the current object type to the type stored in the cache variable. If they agree, the cached feature value is used. The hash table lookup is done only when they disagree. With a "cache hit", the extra cost is just that of retrieving the object tag and comparing it to the stored type. The study in [7] found that the hit rate for dispatch caches was about 80% in the compiler and that this was high enough to have a significant effect on reducing execution time.

3.2 Parameterized Classes

One central feature of the Sather design is the use of parameterized classes. These are classes with one or more type parameters whose values are specified when the class is used. For example, the array class is declared as `ARRAY{T}`. When used, however, the type variable `T` is instantiated to the type contained in the array. Thus, `ARRAY{INT}` declares an array of integers and `ARRAY{STR}` declares an array of strings. Parameterization supports a very common and important form of reuse. In conjunction with Sather's strong typing it causes no increase in performance overhead. To achieve this high performance, we decided to generate separate code for each instantiation of the parameters

of a parameterized class. This allows the code to compile in the targets of calls. The increase in the size of the code does not appear to be substantial. Typically there are a few small classes such as `LIST{T}` which cause the generation of many instantiations, but most classes are not so replicated. The compiler determines the minimal set of self-consistent instantiations required. It also recognizes instantiations which are only used for class inheritance and doesn't generate corresponding code.

3.3 Arrays

Many of the most important structures used in the Sather libraries are based on arrays (see section 4). It was therefore essential to support efficient array access. A very common (but horribly kludgey) paradigm in C programming is to define C structures whose last element is declared to be a one element array. When the space for the structure is actually allocated, however, extra space is included off the end. Even though the final array variable in the structure was declared to have only a single entry, larger indices are used to index into the extra declared space. This allows one to define dynamically sized arrays without incurring the cost of indirecting through a pointer stored in the structure. While this construction is not very clean in C, it is so useful that we made it the basis for Sather arrays. Objects whose defining class inherits from the class `ARRAY{T}` include a dynamically sized array portion at the bottom of the object layout. Such objects also include an attribute named `asize` which gives the number of allocated entries. When a Sather program is compiled, a flag may be set to enable runtime checks that array indices are within bounds. Multi-dimensional arrays are similarly defined and maintain a set of pointers to appropriate subarrays to avoid multiplications on access.

3.4 The Compiler

The Sather compiler is itself written in Sather. The implementation is described in detail in reference [12]. It is a fairly large program with about 30,000 lines of code in 183 classes (this compiles into about 70,000 lines of C code). As such, it provides a very nice example on which to test out concepts for structuring large programs. Because modern workstations have lots of memory and this trend is likely to improve even further in the future, the compiler does not use files to store intermediate structures. Our view is that the virtual memory system is better able to determine what should be resident in memory than the compiler.

The compilation process is directed by a command file which specifies the locations of the Sather source files, C object files, and includes any special directives for compilation. The compiler scans

and parses all relevant sources to build a complete set of code trees in memory. YACC was used to generate a parser which interacts with a hand written lexical analyzer. Many of the different lexical categories have corresponding classes and code trees are generated by building syntax trees out of these objects. The main compilation phase consists of a series of code walks which successively transform the trees into a form suitable for the output of C code. Initial phases implement inheritance and instantiation of the type parameters in parameterized classes. Tables are built to represent the types of all expressions and type checking is performed. A final code walk is used to output a separate C file for each Sather class instantiation.

The Sather runtime support includes a conservative mark-and-sweep garbage collector [1]. Sather programs tend to generate far less garbage than is typical for other languages. Because the Sather compiler generates C code, it is fairly easy to port to new architectures which have existing C compilers. For example, to port the compiler (without the garbage collector) from Sparc to MIPS took one night of work.

4 The Libraries

In this section we describe certain principles which have arisen in the development of Sather library classes. The Sather library currently includes several hundred classes. Eventually, we hope to have efficient, well-written classes in every area of computer science. The libraries are covered by an unrestrictive license which encourages the sharing of software and crediting authors without prohibiting use in proprietary and commercial projects. Currently there are classes for basic data structures, numerical algorithms, geometric algorithms, graphics, grammar manipulation, image processing, statistics, user interfaces, and connectionist simulations.

4.1 Modification vs. Creation

One theme which has arisen in several guises is that it is often better for the routines in classes to modify existing objects rather than to create new ones. The most obvious reason is that modification gives better performance. For example, the `VECTOR` class provides both the routines `plus(v:VECTOR):VECTOR` and `to_sum_with(v:VECTOR):VECTOR`. The first routine creates a new vector and sets its entries to be the sum of the vector the call is made on and `v`. The second routine doesn't create a new vector, but modifies the elements in `self` to be their sum with corresponding elements in `v` and returns it. Such a 'modification' routine allows the programmer to avoid creating potentially large objects when it is not necessary. The behavior of `v1.plus(v2)` can also be obtained

with `v1.copy.to_sum_with(v2)`.

A more fundamental reason for preferring modification to creation arises in the presence of multiple inheritance. Imagine we have an object `c` whose type `C` inherits from both `A` and `B`. The attributes of `c` consist of both the attributes defined in `A` and those in `B`. If all of the routines in `A` and `B` work by returning new objects rather than by modifying existing objects, there will be no way to use them to get `C` objects with both the `A` and `B` attributes filled in. If they work by modifying existing objects, then `c` can be modified first by an `A` routine and then by a `B` routine, thus setting both portions.

Another advantage of modification occurs when several (possibly unknown) objects point to the existing object. If we modify it, they will all see the new entries. If we create a new object, there is no simple way for the other objects to see the new information. Extra indirection would be needed to share information. Modification is also related to a style of object creation in which new objects are formed by duplicating existing objects (which may have undergone several stages of modification) rather than by calling a creation function.

We have often found it useful for a routine which modifies an object to return that object. This allows a series of routine calls to be concatenated in an expression. For example, in the `VECTOR` class one might call:

```
v:=VECTOR::new.to_uniform_random.to_sum_with(v).scale_by(.5);
```

This approach is also used in file output:

```
OUT::s("Variable number ").i(num).s(" is ").r(val).nl;
```

which might produce the output "Variable number 5 is 2.5". In the next two sections we will describe its use in situations in which objects may be resized as a result of a modification.

4.2 Amortized Doubling

A second common motif arises in classes which must maintain dynamically sized information. Common examples include strings, lists, and hash tables. For these structures the size of the information that is to be stored in them is typically not known when they are created. The strategy used in several library classes is to initially allocate a fixed-sized array. Each time the array becomes full, a new array twice as large is allocated. Elements in the old array are copied to the new array. If we make n insertions, we end up with $1 + 2 + 4 + 8 + \cdots + n < 2n$ copying operations. This approach therefore has a cost per insertion of only 2 copies when amortized over a sequence of insertions. It is

also quite space-efficient. Only $log(n)$ chunks must be allocated and garbage collected and the total space used is only twice that which would be required if the size were known in advance.

4.3 Strings

Three of the most basic classes in the library are based on this amortized doubling idea. Sather strings are just arrays of characters in which the characters following the end of the string are 0. This makes the character portion of the string into a legal "C" string which facilitates communication with C programs. Because Sather strings also store the size of the allocated space (as do all Sather arrays), the amortized doubling technique may be used to efficiently build up strings by repeated appending without having to declare the amount of space required in advance. The length of a string is quickly determined by a binary search for the terminating zeros. The string class has routines (with single-letter names) which append string representations of each of the basic types. Such an append may expand the size of the string and so must be reassigned to the source. If `str` holds a string, then a statement to append to it might look like:

```
str:=str.s(" plus ").i(value).s(" makes ").i(sum).s(".").nl;
```

If `str` holds "The value 7" then after appending it might hold "The value 7 plus 8 makes 15.".

4.4 Lists, and Gap Lists

Perhaps the most used class in Sather programming is `LIST{T}`. Objects of this type are used as generic container objects in the same way that Lisp uses linked lists. `LIST{T}` is a parameterized class for holding objects of type `T`. It consists of a variable sized array and an attribute called `size` which specifies how many of the array elements are currently filled. The primary means for adding elements is to append them to the end as in a stack. The size of the allocated space grows by amortized doubling. There are several advantages over linked lists. In a `LIST{T}` each element is directly accessible by its array index rather than having to walk along links. As described in 4.2, many fewer objects must be allocated and garbage collected for `LIST{T}` than for linked lists. It is also much cheaper to perform a "deep copy" or to save and restore such a structure from disk. These advantages hold especially when the lists are large.

One advantage of linked lists is that they can efficiently add and delete elements from the interior of the list. In the Sather libraries we typically use a class called `GAP_LIST{T}` when this functionality is needed. These structures are also based on dynamic arrays, but they use the entire array and maintain the size and location of a "gap", which consists of unused array entries in the middle of the

list. The retrieval routines provide indexed access as if the gap were not present. When elements must be added or deleted, the gap is first moved to the location where the change will occur. This structure (sometimes called "double stack") has been used in several text editors to hold text files. In typical applications successive modifications tend to occur near one another. In this case, the amortized cost of moving the gap is small per insertion or deletion.

4.5 Hash sets and maps

Two abstractions which are used extensively in the compiler and other Sather programs are sets of objects and mappings between objects based on hash tables. Sets of objects are maintained by hashing on the addresses of the stored objects and efficiently support set operations such as union, intersection and set equality. Mappings between objects are represented by tables which pair each object with its image under the mapping.

There is a large literature on the choice of hash functions and collision resolution schemes with good theoretical properties. We have found that we obtain the best performance by using an extremely simple function and resolution scheme. We use the extremely simple hash function which just extracts the lower bits of an object's address (excluding the lower two 0 bits) to index into a power of two sized table. The costs of integer multiply and modulo on machines like the Sparc are so high that extra collisions induced by the simple function are more than compensated for.

The hash table itself is a dynamically sized array. Much of the literature aims at maximizing the "load factor" which is the percentage of the array which contains entries. We keep our hash tables at a load factor of only .5 which allows us to get excellent performance with simple algorithms. It is often recommended that hash buckets be implemented by linked lists. We instead use the simple strategy of putting new entries into the first available slot in the hash table after the hashed value. This approach is susceptible to entries "piling-up" in certain regions of the table. The speed of modern machines at searching through successive entries in an array more than makes up for the cost of avoiding more complex structures. A sentinel element is used to make the inner search loop very efficient. The simple insertion scheme also allows us to directly delete elements without using "delete tags". Each of these choices contributes to tables which are extremely fast and efficient in practice.

4.6 Cursors

There is a cursor class for each container class in the Sather library, such as the lists and hash tables described above (e.g. `LIST_CURSOR{T}` for `LIST{T}` and `OB_HASH_SET_CURSOR{T}` for `OB_HASH_SET{T}`). Objects in a cursor class point into the container class. They support the routines `first` and `next` which move the cursor to the first and successive elements. The current element is retrieved with a call to `item:T` and `is_done:BOOL` is used to test for completion. Each of these cursor classes inherits from the class `CURSOR{T}`. This arrangement allows operations which must be performed on all elements in a set to be defined in terms of cursors and so applied to any data structure which supports them. Examples include filtering out certain elements, combining all elements (e.g. summing all integers stored in a set), and applying an operation to each element.

Cursors can also provide more complex functionality in specific classes. For example, the Sather approach to extracting information from strings is to use the `STR_CURSOR` class. This points into a string and provides information about what is next in the string. Routines are provided that skip over space and extract words, lines, integers, reals and other textual entities. This approach is far more flexible than the "scanf" approach in C because the type of each entity may be tested at runtime as opposed to being specified in a template. The same idea may be used to cleanly encapsulate regular expression search. A special cursor is constructed for each regular expression and is used to scan down strings looking for matches.

Cursors may also be used to order computations. A very common situation is to have dependency relations between a set of computations that must be performed. These relations which constrain the ordering of the computations form a graph. One wishes to perform the computations in an order determined by a topological sort of the graph. Instead of re-implementing this abstraction in each situation in which it arises, we may use a cursor class which implements this abstraction. The dependencies are registered with the class and successive objects are retrieved from it in the proper order. Other examples include retrieving the elements of a tree or graph in depth-first or breadth-first order. Another example which commonly arises in routines for the construction of parsers is to propagate changes among a set of objects until nothing changes. Again a cursor class naturally captures the desired abstraction.

4.7 Graphs

Graphs are an extremely important abstraction throughout computer science. For example, many operations performed by a compiler may be naturally thought of as constructing and acting on certain

graphs. It is therefore important to have library classes which implement a clean graph abstraction. The obvious approach would be to represent graph vertices by objects with an attribute which contains the edge list for that vertex. Each vertex object would inherit from a `GRAPH_VERTEX` class which would contain the routines for manipulating graphs. Unfortunately, this simple approach quickly runs into severe problems. Many typical graph operations take one graph as input and produce another graph as output. An example is the transitive closure operation which might be used in a compiler to determine which routines are reachable by calls from which others. We are often interested in both the original graph and the new graph. In the simple representation, however, these two graphs would have the same vertex objects. Each object can only have a single edge-list, however, and so cannot be a part of two graphs.

We use a different representation which we have found to be very useful in a variety of circumstances. The natural view of an object is as a mapping from attribute names to attribute values. In this view an object is naturally representated by a chunk of memory and attributes are offsets into this chunk. An alternative view is that an attribute name is a mapping from objects to attribute values. The natural representation for this view uses a hash table for each attribute name to map objects to values. This second perspective is more natural in representing graphs. A graph is a hash table which maps from vertex objects to their edge lists. The object itself has no knowledge of which graphs it belongs to and so may belong to as many as desired. Different graph objects may map a vertex object to different edge lists. An operation like transitive closure takes one graph object as input and produces another one with the same vertices as output. The same idea is used in several places in the Sather libraries. For example, the `UNION_FIND` class maintains sets of sets which support efficient set union and element find operations. This shift of representation is useful in a variety of other circumstances as well.

4.8 Vector maps

Other important points about designing powerful class abstractions are illustrated by the classes which represent mappings from one vector space to another. Examples of such mappings are those produced by connectionist networks, linear least-squares fitters, and statistical non-linear regression techniques. In the Sather library, each of these classes inherits from the class `VECTOR_MAP`. This class defines the interface for such operations as determining the dimension of the input and output spaces, computing the image of a vector under the mapping, and miscellaneous operations such as computing the mean square error for a set of training examples.

Once such an abstraction is defined, there are a whole set of "functorial" combining classes which may be defined to construct more complex maps. For example `COMPOSITION_VECTOR_MAP{M1,M2}` represents the mapping which is a composition of two maps of types `M1` and `M2`. Another class `PRODUCT_VECTOR_MAP{M1,M2}` forms the product of two maps, while `SUBSET_VECTOR_MAP` maps a vector to a permutation of some of its components, and `CONSTANT_VECTOR_MAP` represents a constant output vector. These classes may be combined like tinker-toys to build up arbitrarily complex maps which still obey the defining interface.

4.9 Random number generation

We found a similar approach to be useful in the random number generation classes. We wanted a class `RANDOM` which could produce random samples from a variety of different probability distributions (e.g. normal, binomial, gamma, Poisson, geometric). Such samples are generally produced by manipulating samples from an underlying generator which produces real valued random samples uniformly distributed in the unit interval. There are often differing requirements for such an underlying generator, however. For most applications, speed considerations dominate and a linear congruential generator is sufficient. For certain critical applications, however, we do not care so much about speed but require extremely high quality samples. We therefore structured the library classes so that objects of type `RANDOM` have an attribute of type `RANDOM_GEN` which holds the underlying generator. This is dispatched to retrieve uniform random variates. A variety of basic generators are provided. In addition, like vector maps, a variety of combining classes are provided to construct new generators. Examples include classes which generate new samples by summing the outputs of two generators modulo 1 and classes which form new generators by randomly permuting the outputs of other ones.

5 Future work

In this paper, we have described aspects of the design and implementation of Sather, and how they are related to the goals of achieving code encapsulation, re-usability, efficiency and portability. In addition, section 4 illustrates principles used in the design of library classes to achieve the same goals. The Sather language provides powerful features such as parameterized classes and object-oriented dispatch which are essential to achieving code encapsulation and re-usability. The language is small, simple, and efficient. As we continue to actively develop classes and language tools, important new class abstractions become apparent. The current distribution includes X windows user interface

classes and a symbolic debugger. We are working on higher-level user interface abstractions and extended language tools such as an interpreter.

Another direction of research is the further extension of the Sather language to a parallel multiprocessor environment. The language "pSather" [3] is still being modified, but an initial version runs on the Sequent Symmetry. Work is proceeding on a version for the Thinking Machines CM-5. As described in [4], the implementation allows a programmer to achieve reasonable speedup without too much "heroic" effort.

6 Acknowledgements

Many people have contributed to the development and design of Sather. We would especially like to thank Subutai Ahmad, Jeff Bilmes, Henry Cejtin, Graham Dumpelton, Richard Durbin, Jerry Feldman, Franco Mazzanti, Heinz Schmidt and Bob Weiner.

References

[1] H. Boehm and Weiser M. Garbage Collection in an Uncooperative Environment. Software Software Practice & Experience pp. 807-820, September 1988.

[2] Brad J. Cox. *Object-oriented Programming, An Evolutionary Approach.* Addison-Weseley, Reading, Massachusetts, 1986.

[3] Jerome A. Feldman, Chu-Cheow Lim, and Franco Mazzanti. psather monitors: Design, Tutorial, Rationale and Implementation. Technical Report TR-91-031, International Computer Science Institute, Berkeley, Ca., September 1991.

[4] Jerome A. Feldman, Chu-Cheow Lim, and Franco Mazzanti. Parallel Sather: Language Design and Application Experience. Submitted to OOPSLA 1992, February 1992.

[5] Richard P. Gabriel, Jon L. White, and Daniel G. Bobrow. CLOS: Integrating Object-Oriented and Functional Programming. *Communications of the ACM*, 34(9):28–39, September 1991.

[6] A. J. Goldberg and D. Robson. *Smalltalk80: The Language and Its Implementation.* Addison-Wesley, Reading, Mass., 1983.

[7] Chu-Cheow Lim and Andreas Stolcke. Sather Language Design and Performance Evaluation. Technical Report TR-91-034, International Computer Science Institute, Berkeley, Ca., May 1991.

[8] Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall, New York, 1988.

[9] Bertrand Meyer. *Eiffel: The Language.* Prentice Hall, New York, 1992.

[10] Stephen M. Omohundro. The Sather Language. Technical report, International Computer Science Institute, Berkeley, Ca., 1991.

[11] Heinz W. Schmidt and Stephen M. Omohundro. Clos, Eiffel, and Sather: A Comparison. ICSI Technical Report TR-91-047, to appear in "The CLOS Book", edited by Andreas Paepcke, February 1992.

[12] Chu-Cheow Lim Stephen M. Omohundro and Jeff Bilmes. The Sather Language Compiler/Debugger Implementation. Technical report, International Computer Science Institute, Berkeley, Ca., 1992 (in preparation).

[13] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Mass., 1986.

[14] David Ungar and Randall B. Smith. SELF: The Power of Simplicity. In *OOPSLA 1987 Conference Proceedings*, pages 227–241, 1987.