INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center St. • Suite 600 • Berkeley, California 94704-1198 • (510) 643-9153 • FAX (510) 643-7684

An Analysis of the Divergence of Two Sather Dialects

David Stoutamire, Wolf Zimmermann, and Martin Trapp

TR-96-037

August 1996

Abstract

Sather is an object oriented language designed to be simple, efficient, safe, and non-proprietary. It was originally envisioned as a "cleaned-up" version of Eiffel, addressing perceived failures in simplicity and efficiency. The first public implementation (Sather 0) was first released to the public by ICSI in 1991. Shortly after, a compiler group at the University of Karlsruhe created the first native code compiler.

A major effort then began to redesign the language to correct shortcomings in Sather 0 and to make Sather suitable for general-purpose, large scale programming. In part because each compiler group was building a compiler for a moving design target, the two parallel efforts resulted in two dialects, Sather 1 and Sather K. This report analyzes the essential causes of the differences, which result from differences in each group's goals.

1 Introduction

Sather is an object oriented language designed to be simple, efficient, safe, and non-proprietary. Sather was originally developed at the International Computer Science Institute (ICSI), a research institute affiliated with the computer science department of the University of California at Berkeley. The Sather language gets its name from the Sather Tower, the best-known landmark of the campus. The name 'Sather' is a pun - Sather was originally envisioned as an efficient, cleaned-up alternative to the language Eiffel. However, since its conception the two languages have evolved to be quite distinct.

Sather is still growing. The initial Sather compiler (for 'Version 0' of the language) was written over the summer of 1990. ICSI first made a compiler publicly available in 1991. Then, in spring 1992, the group under the direction of Gerhard Goos at the University of Karlsruhe created the first native code compiler for Sather 0, additionally strengthening the type system. This compiler was used in undergraduate programming courses.

After Sather 0, ICSI began a major effort to redesign the language to make Sather suitable for general-purpose, large scale programming outside of the language community. Sather 1 greatly expanded the language, introducing bound routines, iterator abstraction [3], proper separation of typing and code inclusion, contravariant typing, strongly typed parameterization [7], exceptions, stronger optional runtime checks and a new library design. The first Sather 1 compiler was released by ICSI in 1994.

The Karlsruhe group also extended the language. Many of the Sather 1 changes were adopted, but the resulting language was in some respects closer to Sather 0 and does not coincide with Sather 1. The Karlsruhe dialect of Sather came to be called Sather K.

This report attempts to analyze the fundamental reasons for the differences between Sather 1 and Sather K, and originated as a summary of talks between the two groups trying to understand what would be necessary for the dialects to converge in the future. A goal of this work is to make it easier to understand the individual motivations of the two groups so that future discussions will not be bogged down arguing about symptoms when there are fundamental causes that must first be resolved.

In general this report assumes familiarity with at least one of the dialects. It is not intended as a porting guide, and deliberately ignores most minor issues such as syntax. More information about the two languages is available at their web sites:

```
http://www.icsi.berkeley.edu/~sather
http://i44www.info.uni-karlsruhe.de/sather
```

In this report *S1* and *SK* are used to denote the Sather 1.1 language design and the Sather-K design. *S2* is used to refer to a future convergence between S1 and SK. At some places, recommendations for S2 appear, which represent the mutual best judgement of the authors at the time of writing but should not be construed as necessarily representing the views of either the S1 or SK group.

2 Main Goals

Although many of the goals of S1 and SK were the same (eg. to be as efficient as C as well as cleaner and safer than Eiffel), each group had other goals not shared by both.

Some independent goals led to developments by one group that do not necessarily conflict with language goals of the other group because there is no overlap (eg. S1's support for interoperability with other languages). This report does not attempt to cover such unilateral developments, but is instead interested in the goals that precipitated conflicting language designs.

Some unique goals of S1 were:

- Support for separate compilation and multiple developers.

  Issue: Composability of code requires that it be possible to statically check classes over all possible parameterizations.

  Choice: This lead to S1 parameter type bounds and the restriction that overloading may not occur between two unrelated abstract arguments (see section 6).

- Trivial porting to a wide range of platforms.

  Issue: Sather should be available on all platforms. This includes embedded control, the bulk of installed CPU use today. This can be achieved by a custom back end for each platform, if the manpower is available.

  Choice: Because ICSI has limited interest in developing compiler back-ends, S1 chose to use C as a portable assembler. Compiling through C is slow. This intensified the demand for separate compilation, and may have affected the choice for S1 arrays, see section 7 below.

- Extendability for threaded and distributed parallel programming.

  Issue: What about parallel programming?

  Choice: ICSI has been actively engaged in research on parallel and distributed programming since the inception of Sather. ICSI's vision of parallel programming is multithreaded and allows explicit distribution. Multithreaded code imposes very different constraints than data parallel code, which in turn reflects on decisions about the semantics of serial code. (Eg. What are the semantics of a stream shared by more than one thread?)

Some unique goals of SK were:

- Rapid compilation.

  Issue: Karlsruhe has a substantial investment in compiler research, including quality compiler generation tools. Development of SK naturally used these.

  Choice: Because the resulting compiler is so much faster than compiling through C, SK has not been concerned with separate compilation on the class level.

- Clear language design.

Issue: Among others, SK was envisioned as a language with which one would teach students. One wants to be able to expose language constructs in a clean order, without forward references in the language to concepts students are not prepared for. The language is used in undergraduate as well as graduate courses. It is used for teaching both imperative and object-oriented programming [2].

Choice: Some SK constructs differ from S1. For example, *break*! in S1 is *break* in SK, which avoids explaining about iterator notation when loops are first introduced. Similarly, in SK local variables must be declared at the beginning of the block in which they appear.

- Programming in the large.

  Some observations in constructing reliable libraries influenced SK. One goal was to avoid errors (either compile-time or run-time) which point into library code. This led to the introduction of type bounds with structural conformance. For the motivation of structural conformance, see section 6.

3 Determinism

Sometimes the textual or temporal ordering of language constructs is significant, eg. in the evaluation of arguments or constant initializations with side effects. In general, ordering presents problems because programs which depend on a particular ordering are usually relying on some subtle property that would be better if it were explicit. S1 and SK agree that programs which rely on ordering are often poorly designed, and writing such programs should be prevented if possible. The best of all worlds would be to detect reliance on ordering and make it a compile time error. Unfortunately, doing so does not appear tractable.

In general, the S1 approach has been to avoid harm from ambiguous orderings by defining a canonical result. More generally, S1 is deterministic both within and between platforms, with the exception of converting pointer values to integers for use in hash tables, and code which exploits the number of bits in INT or CHAR. SK has chosen the traditional approach which declares programs that depend on nondeterminism to be illegal. However, nondeterministic semantics may offer more opportunities for code optimization.

There are a number of places where the handling of ordering creates a language difference:

- Observed side-effects in shared attribute initialization.

Issue: Both S1 and SK forbid cycles in initialization expressions but do not define a strong order of evaluation. Hence the order of observable side-effects in initialization expressions is not defined.

Choices: S1 restricts shared and const initialization by recursively defining constant expressions, which forbid observable side-effects. This make the order of evaluation irrelevant. SK does not forbid side-effects.

- Value of unassigned variables.

Issue: What should be done with unassigned variables, either local variables or attributes of objects? Setting them to a canonical value allows code which implicitly depends on this value, whereas not defining the value allows code which may run on one compiler but break on another. General checking for use of unassigned variables is presumed to be too expensive.

Choices: S1 defines all unassigned variables to have a default *void* value, which usually amounts to *zero* in the data type. This is desirable for pointer types for garbage collection and is standard practice in many collected language implementations. SK defines *void* as a value that denotes the absence of an object, which makes sense only for reference types. The SK compiler (pessimistically) warns on uninitialized variables.

Observation for S2: Perhaps S2 should distinguish between sentinel values and unassigned variables, since sentinal values (eg. NULL in C) can be algorithmically convenient.

- Rules for resolving conflict during code inclusion.

Issue: What should be done with multiply defined class features? Multiply defined features are either deliberate or a mistake. The former should be easy to express while the latter should be a detectable error.

Choices: S1 distinguishes between features textually defined in a class and those included from another class. Order of definition and inclusion is irrelevant; features defined in a class always override those defined out of the class, and conflicts between features defined out of the class must be resolved explicitly at the point of inclusion. In SK, textual order of definition is significant; later definitions override earlier ones. SK made this choice because of its simplicity, and because an order is required for SK aggregate expressions (see below and section 4).

- Ordering of attributes.

Because class features are ordered, SK uses a textual order of object attributes which is important to the definition of the SK *aggregate* expression. The aggregate expression is motivated by SK's treatment of encapsulation (see section 4—there is no equivalent in S1.)

- Evaluation of arguments.

Issue: How are routine arguments evaluated? How are out arguments assigned - what happens when they are aliased? If an order of evaluation is defined, optimization must operate under more constraints; if an order is not defined, programs may break when switching compilers. The quantitative and qualitative aspects of either choice are not well researched in the compiler literature nor even in the compiler community.

Choices: S1 defines a canonical left-to-right order of evaluation of all arguments. SK makes it implementation dependent. The motivation for this choice is the better efficiency of the generated code. There is historical precedent for leaving the order undefined.

Observation for S2: many of these ordering issues would be resolved by a way to distinguish expressions with side effects from those with none.

- Restrictions on **case**.

Issue: What is permitted in the **case** statement? Some languages restrict the target values to be constants which can be evaluated by the compiler (C **switch**), but other have defined a sequential semantics (Lisp *cond*). Restricting it to constants guarantees ordering can't matter.

Choices: S1 defines the case statement as syntactic sugar for an **if-then-elsif** chain. However, the compiler exploits sequences of constants whenever they occur. SK requires the targets to be constants.

4 Encapsulation

There are two major differences of interpretation of encapsulation between SK and S1.

S1 interprets **private** per-class; that is, code may invoke a private method on another object of the same class. SK interprets **private** per-object; it is not legal to dot into private features. In SK, private features may only be accessed implicitly via *self*. In this sense, SK has a stronger notion of privacy than S1.

S1 requires that the creation and initialization of an object execute code in the object's class, and that access to private attributes by code outside of the class be strictly impossible, even by *pickling* - converting entire data structures to and from a common format such as ascii files for persistence. In this sense, S1 has a stronger notion of privacy than SK.

The implications of these different interpretations are far-reaching:

- Creation of objects.

Issue: How are objects created?

Choices: In S1, an expression *new* may be used to create an object of the class in which it appears, which may then have its attributes filled in. The standard S1 idiom is:

```
create(a, b : FOO) : SAME is
    res ::= new;
    res.attr1 := a;
    res.attr2 := b;
    return res
end;
```

Because such routines are so common, S1 provides syntactic sugar # for calling *create*. This sugar also provides type inference. Because *new* is only effective within the class, it isn't usually possible to create uninitialized objects outside of the class.

In the S1 idiom, the *create* routine creates a new object and then fills in the attributes, but SK can't do this if the attributes are private. Instead, in SK, # creates a new object, where all attributes are initialized according to the initialization expressions of their declarations or to the default values, if no explicite initialization is given.

For convenience there are additional operators in SK. ## creates an object of the same type as an argument. Furthermore the *aggregate* expression allows filling in of public attributes in the textual order they were defined in the class. If class T has two attributes *attr1* and *attr2* defined in that order, $\#T\{a, b\}$ creates an object of class T where a is assigned to *attr1* and b is assigned to *attr2*. Alternatively, this can be made explicit by $\#T\{attr1 := a, attr2 := b\}$.

- Copying objects.

  Issue: How are objects copied?

  Choices: In S1, there is no special language support for copying; it is done in the same way as object creation. This prevents copying objects without the class getting control, for example to assert unique object ids. In SK, a special predefined method *copy* that makes a new object with the same attributes as *self* is automatically generated by the compiler for every class. *copy* may be redefined by the user.

Observation for S2: a compromise that would allow the per-object privacy of SK with the simplicity and strong encapsulation of S1 would be to somehow distinguish special methods such as *create* which would automatically have a new object as *self*.

5 Abstract classes

Much has been made of the difference between S1 and SK abstract/partial classes, but they are very similar. The essential distinction is that in S1, $ is part of the name and in SK it is an interface-extracting operator. To translate code written using one into the other:

1. To implement S1 abstract classes in SK, leave every method deferred.
2. To implement S1 partial classes in SK, leave every method deferred. SK **is deferred** is equivalent to S1 **stub**. SK abstract classes may not be instantiated.
3. To implement SK abstract classes in S1, make an abstract class with the signatures and another class with the code.
4. Where SK **subtype of** is used, in S1 use < $ as well as **include**.
5. Where S1 < is used, use in SK **subtype of**.

6 Compositionality

Both S1 and SK provide reference classes (ordinary, heap-allocated objects) as well as value classes (usually kept in registers or on the stack because they are immune to aliasing.) In SK each user-defined class must be either subtype of VALUE (indicated by the keyword value) or subtype of REFERENCE (default). Because SK does not allow supertyping, it isn't possible to have user-defined supertypes of both VALUE and REFERENCE as is possible in S1. Because S1 requires that an instance of a type parameter must be a subtype of type parameter bounds, forbidding common supertypes of VALUE and REFERENCE would require a different version of a class for reference typed parameters and value typed parameters. In SK, the problem is solved by *structural conformance*, i.e. the instance of a generic parameter must conform to its type bound, but need not to be a subtype. This is why supertyping is not needed in SK.

Structural parameter bounds are further motivated by the following scenario. Users should be able to use libraries by multiple independent developers without changing the source of the libraries. Suppose a library L_1 contains a generic class $A(T < Y)$ and the class defining type bound Y. Suppose further that library L_2 contains class X which structurally conforms to Y. Because of structural conformance in SK it is legal to instantiate T with X, i.e. to write $A(X)$. In S1 this is only legal if before either Y is declared as a supertype of X or X is declared as a subtype of Y. Hence S1 requires changing L_1 or L_2 or introducing additional code to allow the expression $A(X)$.

In general, SK structural parameter bounds can be translated to S1 by introducing an appropriate abstract class. SK parameter bounds allow the parameter to be structurally conformant instead of requiring a subtype. This implies that in a class $C\{T < \$A\}$, a T can not be assigned to a $\$A$ because it might not be a subtype. S1 parameter bounds allow code which is not legal in SK, and S1 classes parameterized this way would need special handling to be expressed in SK. For example:

```
class FOO{T < $A} is
    bar(arg : T) : $A is return arg end
end
```

has no simple translation in SK. In SK all requirements on generic parameters are derived from the class body and checked upon instantiation of the generic parameter.

One motivation of structural parameter bounds was to make it possible to use libraries written by multiple independent developers. For the same reason, S1 prohibits some forms of overloading which SK allows. S1 forbids overloading between two abstract types with no subtype relation between them, while SK allows it. For example:

Developer L_1 writes:	Developer L_2 writes:
foo(*arg* : $\$A$)	**class C < $A, $B is** ...
foo(*arg* : $\$B$)	

This code is illegal in S1 unless $\$A$ subtypes from $\$B$ or vice versa.

There are times it is convenient to have such an overloading when $\$A$ and $\$B$ are not anticipated to have a common subtype. Note that $\$A$ and $\$B$ are not defined in either L_1 or

L_2, but in some other library on which both L_1 and L_2 depend. In this example, L_1 and L_2 cannot be compiled together, although each compiles independently. When allowing such overloading, there is the implicit constraint that no class may be created which subtypes from $\$A$ and $\$B$ at once. Neither S1 nor SK can express this constraint, but S1 chooses to disallow it; L1 and L2 cannot be composed unless the type system guarantees that any subtypes can be disambiguated with a most specific type. SK allows such overloading in class declarations, as long as overloading resolution is unique for the program being compiled.

7 Arrays

There are two essential differences between S1 and SK arrays. The functionality can be seen to be the same, although the way some uses are expressed can be seen to be quite different; for example, SK builds in higher-dimensioned arrays, while S1 constructs them as ordinary library classes.

1. S1 only allows a single contiguous primitive array portion, obtained by including *AREF* (or *AVAL*). Higher dimensional arrays and objects with multiple conceptual array portions are then constructed as library classes.

   Instead of including a distinguished class, SK uses inline array attributes. That is, attributes of array type can be contiguous with the other fields of an object, while S1 requires an indirection to a separately allocated array object if there is more than one. (S1 chose this approach because S1 immutable arrays have constant size. This constraint was inherited from C's equivalent restriction on structs.)

2. In SK, there are static value arrays *ROW*[], dynamic value arrays *ARR*[], and flexible reference arrays *ARRAY*[]. Static value arrays *ROW*[] allow only one dimension. In the case of *ARR*[] and *ARRAY*[], the dimension is part of the type. Inside the square brackets the sizes of the single dimensions are given. These are required in type constructors, elsewhere they can be given as '*'. For the type, only *ROW*[] distinguishes the value of those constants. E.g. *ROW*[*3*] and *ROW*[*4*] are different types, while *ARR*[*3*] and *ARR*[*4*] are equal types.

8 Iterators/Streams

Where S1 has *iterators*, SK has *streams*. In both languages these can be bound like routines and are then called iterator closures or bound streams respectively. Besides *direct stream calls* SK has *stream objects*, too. The former behaves like S1 iterators. The latter allowes stream state to be handled like any other value in the language. E.g. the termination status may be checked, stream state can be passed to other methods or can be resumed in a later loop or outside of loops.

9 Handling of built-in methods

In S1, there are no predefined methods in every class. In SK, there are a variety. In general, the S1 policy was to place suspicious methods in the *SYS* class so they would be easily noticed (as in Oberon).

Built-in Methods:

- *string*: In SK, the predefined *string* dumps all fields of the object, including private attributes. However, it may be redefined. In S1, *str* must be defined by the class as an ordinary routine; a debugger or reflective methods in *SYS* must be used for getting at private attributes. In SK *string* is often used for debugging.
- *is_equal*: In SK, the predefined *is_equal* computes object equality for reference types and recursively for all attributes for value types. However, it may be redefined. = is sugar for *is_equal*. In S1, *SYS* :: *ob_eq* does exactly the same thing, and containers that distinguish between object and value equality do so by typecasing on *$IS_EQ*. In S1, = is sugar for *is_eq*.
- *type*: In SK, the predefined *type* returns a type, which may be converted to a string by application of *string*. In S1, this is accomplished with *SYS* :: *type* and *SYS* :: *str_for_type*.
- *copy*: In SK, all reference objects support a built-in copy, discussed above. It is possible to redefine *copy*.

This list is not exhaustive.

Observation for S2: both languages manage to accomplish the same things in slightly different ways, with the exception of converting pointers to hashable integers (important for performance).

10 Miscellaneous

There are many language differences that do not appear to fit in one of the above catagories, and are listed here for future discussion. These differences may be ascribed to poor communication, different timings of the language design, or other goals not mentioned here.

- Value (S1.1: immutable) semantic types cannot be aliased in either S1 or SK. S1 prevents this by (conceptually) forcing a copy on modify, while SK allows modification in place but copies on assignment (like C structs).
- S1 disallows implicit disposal of return values. SK allows it.
- SK has implicitly defined *res* while S1 does not.
- SK allows implicit coercion (for example, between ints and floats). S1 does not. Implicit coercion is dangerous because precision can be silently dropped.

  Recommendation for S2: allow implicit coercion only where precision may not be lost, or disallow it.

- SK defines many kinds of syntactic sugar for ascii characters that S1 doesn't try to use. Some are not compatible with S1 sugar, such as S1's *negate*.
- In SK, most errors turn into catchable exceptions. In S1, most aren't. The question is what acceptable semantics are for production code, where events that could raise exceptions that would be caught are deliberately not checked because the user wants maximal efficiency.
- S1 disallows multiple types in when clauses of typecase statements to avoid textual ambiguity when there are iterators. SK allows multiple types.
- SK has dynamic constants, which are variables that may only be assigned at declaration. S1 has no equivalent feature. The S1 **const** is a **shared const** in SK. Similarly, SK calls **const** what S1 calls **once**.
- The base library classes are very different. In SK there are *SHORT_INT* and *LONG_INT* classes, *Inf* and *NaN* are keywords, instead of *FSTR* there is *STRING*, SK has a *TEXT* class, I/O is handled differently, and there are many other changes.
- The S1 and SK exception mechanisms are very similar, except that in SK, the raised type must subtype from *EXCEPTION*.

  Recommendation for S2: we should consider making raised types part of signatures as is done in Modula 3.
- S1 uses a different syntax for closure and closure types than SK's bound routine and bound types.

References

[1] G. Goos, *Sather-K, The Language*, TR-95-8, University of Karlsruhe, Falkulty of Informatics, 1995.

[2] G. Goos, *Vorlesungen über Informatik II*, Springer Verlag, 1996.

[3] S. Murer, S. Omohundro, D. Stoutamire, C. Szyperski, "Iteration abstraction in Sather", *Transactions on Programming Languages and Systems*, Vol. 18, No. 1, p. 1-15, 1996.

[4] S. Omohundro, *The Sather Language*, International Computer Science Institute, 1991.

[5] S. Omohundro and C. Lim, *The Sather Language and Libraries*, TR-92-017, International Computer Science Institute, 1992.

[6] D. Stoutamire and S. Omohundro, *The Sather 1.1 Specification*, TR-96-012, International Computer Science Institute, 1995.

[7] C. Szyperski, S. Omohundro, S. Murer. "Engineering a programming language: The type and class system of Sather", In Jurg Gutknecht, ed., *Programming Languages and System Architectures,* p. 208-227. Springer Verlag, Lecture Notes in Computer Science 782, 1993.