pSather: Layered Extensions to an Object-Oriented Language for Efficient Parallel Computation

Stephan Murer* Jerome A. Feldman†

Chu-Cheow Lim‡ Martina-Maria Seidel§

TR-93-028 (2nd revised edition)

December 1993

Abstract

pSather is a parallel extension of the existing object-oriented language Sather. It offers a shared-memory programming model which integrates both control- and data-parallel extensions. This integration increases the flexibility of the language to express different algorithms and data structures, especially on distributed-memory machines (e.g. CM-5). This report describes our design objectives and the programming language pSather in detail.

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

†ICSI and Computer Science Division, U.C. Berkeley. E-mail: `jfeldman@icsi.berkeley.edu`.

‡ICSI and Computer Science Division, U.C. Berkeley. E-mail: `clim@icsi.berkeley.edu`.

§ICSI E-mail: `mseidel@icsi.berkeley.edu`.

Contents

1 Introduction

The parallel Sather (pSather) project focuses on language and library support for *general purpose parallel programming*. Sather began as a smaller, simpler and faster variant of Eiffel [55]. A preliminary version [60] released in summer 1991 has developed a significant global user community. The design of Sather 1.0 [61], released in 1993, retains much of the original simplicity, but includes several additional features that were found to provide the greatest increase in functionality. These include exception handling, a form of closures, and a general iterator feature [58] that is particularly important for parallel computation. The central theme of Sather is flexible encapsulation: libraries of carefully written parameterized classes support programs that are efficient and powerful and also easy to write, read and reason about. Sather is basically, but not religiously, object-oriented.

The pSather project is research oriented, but does involve running systems and applications. An implementation of our first design has been running on the Sequent Symmetry and Sparcstation since early 1991 and a CM-5 version has been running (and changing) since early 1992. Implementation and performance monitoring of a range of non-trivial problems has been one of the cornerstones of the design. Our thesis is that the *flexible encapsulation mechanisms* that characterize Sather are even more important in parallel computation. The key constructs of pSather include support for both distributed and replicated data structures and a statement for data-parallel computation with associated distributed objects. At a more fine-grained level are uniform memory addressing, placement operators, and general mechanisms for thread creation and synchronization.

We assume that the computers of the forseeable future will have *non-uniform memory access (NUMA)* and that they will incorporate increasing parallelism. While there are reasonable methodologies for programming parallel computers for certain classes of tasks, there is nothing like the parallel equivalent of general programming in languages like C++, Modula-2, Eiffel, etc. Much of the work that attempts to address this problem is actually more suitable for concurrent and distributed computing, which differ in several important ways from parallel computing.

Distributed and concurrent systems are often designed to support multiple parallel tasks which *compete* for shared resources. Fairness and mutual protection are central issues in these systems and consume a large part of the available computing power. The main goal for parallel systems, however, is to make many resources (processors, I/O-devices, etc.) *cooperate* on one large task. Although the interaction and synchronization of multiple flows of control are common issues in both fields, the trade-offs in parallel systems are fundamentally different from those in distributed and concurrent systems. Current attempts to produce low-latency local networks are considered in section 9.6.

True parallelism differs from concurrency in that multiple threads of control are active on the same data. This occurs in shared-memory machines like the Sequent and in the clusters of designs like the DASH [47]. It turns out that the coordination mechanisms needed for shared memory hardware are also important for the shared address space abstraction underlying pSather. Distributed computing is inherently concurrent and also differs in two other crucial ways from parallel computing. A major concern in distributed computing, recovery from partial failures, is not an issue for parallel machines. More importantly for our purposes, the memory latency penalty for remote access is tolerable (~ 100 cycles) in parallel machines but not in distributed networks where the latency can be 10,000 or more times the fastest internal memory access.

Distributed computing is largely concerned with the realization of systems that are inherently separated in space and are best thought of as loosely coupled modules. In parallel computing, the emphasis is on efficient execution; there is no inherent need for the programmer to deal explicitly with multiple threads of computation and most people would rather not. The implicit approach to parallel programming strives for languages and compilers that totally mask all considerations of parallelism. This may eventually encompass general programming, but we seem to be quite far from that. In explicit approaches, such as pSather, the goal is to provide constructs that help in the construction of sound and efficient parallel code. A dilemma that confronts such efforts is that programmers prefer shared-memory models but physics requires that large parallel computers have distributed memory. We offer a *cluster* machine model which reflects the NUMA characteristic but retains a shared-memory programming model.

Our basic goal is to explore how well the flexible encapsulation paradigm of Sather can support general purpose parallel programming. We envision libraries of classes for powerful data structures and operations that the ordinary user can employ without any explicit concern for parallelism. This can be viewed as an attempt to extend data-parallel and functional programming styles to complex data structures such as sets, trees, and hash-tables. Our hope is that the parallel programming constructs of pSather will be used mostly by the designers of these parallel class libraries, but application programmers can also directly employ both control and data parallel mechanisms as needed.

The basic pSather design is grounded in a shared-memory computational model and it is natural to write pSather programs that yield identical results independent of placement. Considerable research has been focused on the development of low-level support for the shared-memory abstraction and, if this works out, this placement-independent style of pSather programming will be an excellent fit. But none of the proposed general shared-memory simulations perform well enough for our needs, so pSather also has features that support explicit placement of objects and of threads. Again, we hope that most of the concern with placement can be confined to class libraries and that the average user can employ parallel data structures obliviously. To the extent that this works out, specialized class libraries can make pSather an implicitly parallel language for various domains. This is, we believe, the best current hope for general purpose parallel programming.

1.1 Roadmap of this Report

pSather adds only a few constructs to Sather, but each construct carries a significant semantic burden. This paper describes the new features and how they interact with Sather 1.0 constructs, and gives examples of their use. There is some discussion of design decisions and much more can be found in [27] and [48]. We can order the features of pSather into four layers as illustrated in Figure 1. Upper layers are proper extensions of the lower layers.

In section 2 we give a brief introduction into Sather 1.0 the base language for pSather. A complete description of Sather 1.0 can be found in [61]. A complete grammar (including Sather) of pSather is in appendix A.

Section 3 adds a second layer to the single processor model of Sather, incorporating a multiple processor shared-memory model. We introduce non-blocking routine calls for spawning parallel threads of execution. Synchronization among parallel threads in pSather is provided by built-in synchronization and communication objects, called *gates*. The features of these gate classes plus the related `lock` statement cleanly support all of the synchronization mechanisms in the literature and also embody our versions of futures and of thread control. This design has been stable since summer 1991 under the old name of "monitor" and is discussed in detail in [27]. Section 3.3 contains a brief description of the gates and how they interact with the rest of the design.

Sections 4–7 present material that is new since [27] and was largely motivated by distributed memory considerations. pSather continues to be based on a shared address space abstraction, but some additional constructs greatly help in mapping programs to distributed memories.

Section 4 describes the third layer which extends the shared-memory model to a shared address space model consisting of multiple shared-memory *clusters*, each with potentially multiple processors. We introduce remote routine calls for moving the locus of execution to remote clusters. We describe the language features that make the two-level structure of the shared address space visible in the programming model. The `with-near` statement helps the compiler by asserting that a variable references only objects located on the local cluster.

Section 5 describes how pSather supports copying and migration of objects among clusters.

In section 6 the built-in class `$SPREAD` is introduced as a basic building block for replicated objects and reduction operations. In contrast to ordinary objects which reside entirely on one

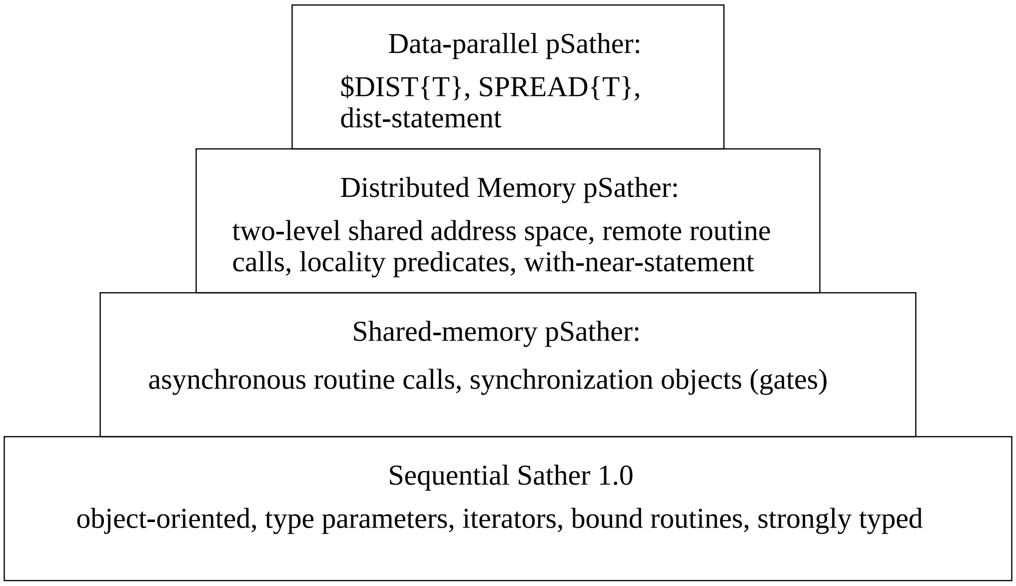

Figure 1: Layered Design of pSather

cluster, `$SPREAD`-objects are spread out on all clusters of the machine.

Section 7 describes the built-in class `$DIST` and the associated `dist` statement for data-parallel computations. These constructs support a high-level way of forking multiple threads, each working with its local data, and make up the fourth layer in our construction. Classes that descend from `$DIST` all share a common mechanism for distributing chunks of data over multiple clusters. It becomes quite natural to write `dist` statements in which the execution of the parallel body is completely distributed and co-located with the appropriate data chunks, such that there are no remote accesses.

We compare pSather to a number of related approaches in section 8, and discuss some possible future directions in section 9. We give our conclusions in section 10.

1.2 Grammar Notation

The grammar rules are presented in a variant of Backus-Naur form. Non-terminal symbols are represented by strings of letters and underscores in an italic font and begin with a letter. The nonterminal symbol on the lefthand side of a grammar rule is followed by an arrow " $\Rightarrow$ " 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.

All code examples and names refering to classes and features in the code are printed in the `typewriter` font.

2 Sequential Sather

In this section we will briefly describe Sather 1.0, the base language for pSather. We will concentrate on the relevant concepts for the purpose of this report. Most of this section consists of condensed version of [61]. Refer to this report for a full definition of the language.

Sather is an object-oriented language that supports highly efficient computation, powerful abstractions for encapsulation and code reuse, a flexible interactive development environment, and constructs for ensuring code correctness. It has garbage collection, statically-checked strong typing, multiple inheritance, separate implementation and type inheritance, parameterized classes, dynamic dispatch, iteration abstraction, higher-order routines and iters, exception handling, constructors for arbitrary data structures, and assertions, preconditions, postconditions, and class invariants. The development environment integrates an interpreter, a debugger, and a compiler. Sather code can be compiled into C code and can efficiently link with C object files.

Data structures in Sather are constructed from *objects*. Each object has a unique *type* which defines the supported operations. Each Sather variable has a declared type which determines the types of objects it may hold. Types are defined by textual units called *classes*. Sather programs consist of sets of classes. Classes may define the following *features*: *object attributes* which make up the internal state of an object, *routines* which perform operations on objects, *iters* which encapsulate iteration abstractions, and *shared* and *constant* attributes which are shared by all objects of a given type. Features may be declared *private* which restricts access to the class in which they appear. Access routines are automatically defined to access and modify object, shared, and constant attributes. The set of non-private routines and iters in a class define the *interface* of corresponding type. Routine and iter definitions consist of *statements* and these in turn are constructed from *expressions*. There are special *literal expressions* for boolean, character, string, integer, and floating point objects. There are also certain predefined classes and features.

2.1 Types and Classes

There are three kinds of objects in Sather: *value objects* are passed by value (e.g. integers), *reference objects* are referenced via pointers (e.g. strings) and *bound objects* are the Sather version of closures. There are five kinds of types: *value types* describe value objects, *reference types* describe reference objects, *bound types* describe bound objects, *abstract types* describe sets of reference types, and *external types* are used to access code from other languages. Variables may be declared by abstract types but there aren't objects of abstract type. Neither variables nor objects may be of external type.

The *type graph* is a directed acyclic graph whose nodes are types and whose edges define *type conformance*. The type graph specifies the object types that a variable may hold and imposes *conformance constraints* on interfaces of types. If there is a path in the type graph from a type *t1* to a type *t2*, we say that *t1* is an *ancestor* of *t2* and that *t2* is a *descendant* of *t1*. Only the abstract types and bound types can have descendants in the type graph.

A type is said to *conform* to each of its ancestors in the type graph. The fundamental Sather typing rule is: "*An object can only be held by a variable if the object's type conforms to the variable's declared type.*" Sather is statically type-safe and it is not possible to write programs which compile and yet violate this rule. Variables declared by value or reference types can only hold objects of the same type. Variables declared by abstract types can typically hold more than one type of reference object and variables declared by bound types can typically hold more than one type of bound object.

There are four kinds of classes: *value classes* which define value types, *reference classes* which define reference types, *abstract classes* which define abstract types, and *external classes* which specify interfaces between Sather and other languages. The bound types do not correspond to classes.

Value, abstract and external class definitions begin with the keywords `value class`, `abstract class` and `external class`, respectively. The most common classes are reference classes and their definitions begin with the single keyword `class`. Class names must be uppercase. The names of abstract classes must begin with a dollar sign `$`.

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

Abstract, reference, and value classes may be *parameterized* by type parameters which are specified when the class is referred to. Type parameter names must be uppercase and may be used within the body of a parameterized class wherever a type specifier is allowed. The semantics of parameterized classes is precisely the same as if there were a separate copy of the class text for each distinct parameter instantiation. The parameters are textually replaced by their specified values in these copies.

Sather types are specified syntactically by *type specifiers* of one of the following forms:

- `A`, `$A`. The name of a non-parameterized class or a parameterized class in which all parameters specify default values.
- `A{B,C}`. The name of a parameterized class followed by type specifiers for its parameters. If all parameters have default values, then the class name alone is also a legal specification.
- `{A}`, `{A,B,C}`. *Tuple types* are built-in value types which are specified by a list of type specifiers enclosed in braces.
- `T` inside `class B{T} is ... end`. The name of a type parameter within the body of a parameterized class.
- `ROUT{A,B}:C`, `ITER{A!}:D`. Bound routine and iter type specifiers consist of one of the keywords `ROUT` or `ITER` followed by optional argument types in braces, followed by an optional return value type. Bound iter argument types may be followed by a "`!`".
- `SAME`. The special type specifier `SAME` refers to the type of the class in which it appears.

The parameters of parameterized classes are specified *positionally* from left to right. A type specifier may specify fewer parameters than are declared in the class if the remaining parameters have default values.

If a type parameter specifies a *type constraint*, then only descendants of that type may be used as values for the parameter. The components of parameter constraint type specifiers may include any of the parameters in the parameter list (eg. `A{E,S<$SET{E}}`) but may not include `SAME`.

The components of parameter default type specifiers may include earlier parameters in the parameter list (eg. `A{E,S<$SET{E}:=HSET{E}}`) but may not include `SAME`. Parameterized class definitions must lead to valid classes when the parameters are assigned left to right by the following procedure: parameters with a default type are assigned that type and the rest are assigned their constraint type if present or `$OB` otherwise.

The set of non-private routines and iters of a class is called the *interface* of that class. The type system constrains the interfaces of classes according to the rule: "*Each descendant of an abstract class must define a routine or iter corresponding to each routine or iter in the abstract class. It must have the same name, the same number and types of arguments, and a conforming return type if one is present and must not have a return type otherwise.*"

Most of the edges in the type graph are defined by the *class_inheritance* clause in class definitions. The first optional portion consists of "`<`" followed by a list of abstract type specifiers. These specifiers may not contain "`SAME`" but they may contain type parameters. A type may not be listed more than once.

This clause specifies that, after any type parameters are specified, the type defined by the class is beneath each of the specified abstract types in the type graph. There must not exist a cycle of abstract classes such that each inherits from the next, ignoring the values of any type parameters. Every type is automatically a descendant of `$OB`.

A second optional portion consists of ">" followed by a list of type specifiers and may only appear in abstract class definitions. It means that, after any type parameters are specified, the type defined by the class is above the listed classes in the type graph. These type specifiers may not contain "`SAME`", be of external type, or be a type parameter (though they may contain type parameters). A type may not be listed more than once. There must not be a cycle of abstract classes such that each is over the next, ignoring the values of any type parameters.

If both a "<" portion and a ">" are present, then each class listed in the ">" portion must have each class listed in the "<" portion as an ancestor using only links defined by "<" clauses. This ensures that conformance can be tested by examining a sequence of classes beginning at one of the two classes in question.

2.2 Features

The main body of each class is a semicolon separated list of feature definitions and `include` clauses. The semantics of a class is independent of the textual order of these class elements. The five kinds of features are: *constant attributes*, *shared attributes*, *object attributes*, *routines*, and *iters*. Each feature has a name and may potentially contribute a "reader" and a "writer" routine of the same name to the class interface. The feature namespace of a class is separate from that of other classes and from the class namespace. If a routine or iter is "private", then it may only be called from within the class and is not part of the class interface.

Classes in Sather may define three kinds of attributes.

- *Object attributes* are variables which are part of the internal state of reference objects. Only abstract and reference classes may define object attributes.

- *Constant attributes* are accessible by all objects in a class and may not be assigned to. If a type is specified, then the construct defines a single constant attribute named *ident*. It must be initialized by the expression *expr* which must be statically evaluable by the compiler.

  Each constant attribute definition causes the definition of a "reader" routine with the same name which returns the constant's value. It has no arguments and its return type is the constant's type. The routine is `private` if and only if the constant is. Constants may not be assigned to.

- *Shared attributes* are like constant attributes but may be assigned to.

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

Each shared attribute definition causes the definition of two routines with the same name. The "reader" routine returns the value of the shared and has a return type which is the shared's type and no arguments. It is private only if the shared is declared "`private`". The "writer" routine sets the value of the shared and has a single argument whose type is the shared's type and no return value. It is private if the shared is declared either "`private`" or "`readonly`".

Routine definitions contain the code associated with classes. Routines may have zero or more arguments, each of a declared type.

```
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)
```

Routines may also have a return value. Within the body of the routine, the local variable `res` is used to refer to this value. When the routine begins execution, `res` is initialized to `void`. When the routine exits, either at its end or due to a `return` statement, the value `res` is returned. Tuple types are used to return more than one value.

The optional `pre` construct contains a boolean expression which must be true when the routine is called. When checking is enabled, the expression is evaluated when routine is entered and an exception is raised if it is false. The expression may refer to both `self` and the routine arguments.

The optional `post` construct contains a boolean expression which must be true when the routine returns. When checking is enabled, an exception is raised if it is false on return. The expression may refer to `self`, the routine arguments, and `res`.

When a routine call is made on a variable of an abstract type, any `pre` and `post` tests defined for the routine in the abstract class will be checked in addition to those of the routine itself. The body of a routine definition is a list of statements.

Abstract classes may define non-private routines without the body "`is` *stmt_list* `end`". Such routines specify an interface without an implementation.

Iters are similar to routines but encapsulate iteration abstractions. Iter names end with an exclamation point `!`. This symbol is part of the name and may not be separated from the rest of it. Iters may only be called within `loop` statements. The type specifiers declaring iter arguments may be followed by a `!` symbol to indicate that they will be re-evaluated on each iteration.

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

The description of routine arguments, return values, and `pre` and `post` constructs applies equally to iter definitions. Unlike routine bodies, iter bodies may include `yield` and `quit` statements. Iter bodies may not contain `return` statements. When checking is enabled, the `pre` clause is evaluated each time the iter is called and the `post` clause is evaluated each time it yields, but not when it quits.

Include clauses: Implementation inheritance is defined by one of two forms of `include` clause. The form with "`::`" is used to include and possibly rename a single feature from another class. The other form includes an entire class but may cause features to be undefined or renamed with "*feat_mod*" clauses. When an abstract class is included, any routines or iters without bodies are ignored. If the "`include`" clause starts with the keyword `private`, then any included feature which isn't modified is made private.

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

The type specified by *type_spec* must not be an external type, a bound type, a type parameter (though type parameters may appear within the type specifier) or any type specifier containing `SAME`. There must not be an "include path" from a reference type to a tuple type or one of the built-in value types: `INT`, `FLT`, `FLTD`, `FLTE`, `FLTDE`, or `BITS` (section 2.6). There must not be an "include path" from a value type to the built-in array type `ARR` (section 2.6). There must not be a cycle of class names such that each named class includes the next named class (ignoring the values of any type parameters).

Each "*feat_mod*" clause begins with an identifier which is optionally followed by a list of argument types and a return type. These type specifiers are interpreted with "`SAME`" taken to be the included type. The included type must define a feature with the specified signature. Iter signatures must mark arguments with a "`!`" if they are so marked in the original definition. Object, shared and constant attributes are described by using the type signature of their "reader" routine. The feature signature is followed by the transformation symbol: "`->`".

If no clause follows the "`->`" symbol, then the specified feature is not included in the class. When the reader routine for an object, shared, or constant attribute is undefined in this way, the corresponding attribute and writer routine are also undefined.

If an identifier follows the "`->`" symbol, it is used to rename the feature. The included definition is textually identical to the definition in the original class, i.e. routines and iters retain their bodies and attributes retain their initializing expression.

When a feature is renamed, the new name may optionally be preceded by either the keyword "**`private`**" or "**`readonly`**". If no keyword appears, then routines and iters become part of the public interface and attributes have both their "reader" and "writer" routines made public. With the keyword "**`private`**" no additions are made to the public interface. With the keyword "**`readonly`**" shared and object attributes have only their "reader" routines made public.

Two routines or iters are said to "*conflict*" if they have the same name, the same number and types of arguments, and either both have or both do not have a return value. 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. Element modification clauses can be used to resolve these conflicts.

2.3 Statements

The body of a routine or iter is a semicolon separated list of statements. Statements in a statement list are executed sequentially unless interrupted by a call of **`return`**, **`quit`**, **`yield`**, or **`raise`**.

2.3.1 Declarations

Declaration statements are used to declare the type of one or more local variables.

```
i,j,k:INT
```

Local variables may also be declared in assignment statements and **`typecase`**-statements. Unlike many languages, the scope of local variables in Sather is the entire routine or iter in which they appear. Local variables must be declared exactly once in each body and the declaration must be the first occurrence of the variable. Local variable names within a routine or iter must be distinct from each other and from any argument names. Local variable names may shadow feature names in the class, however. Local variables are initialized to **`void`** at the beginning of a routine or iter.

2.3.2 Assignments

Simple assignment statements are used to assign objects to locations and can also declare local variables.

```
a:=5
b(7).c:=5
A::d:=5
[3]:=5
e[7,8]:=5
```

```
B::[7,8,9]:=5
_:=f
g:INT:=5
h::=5
```

The expression on the righthand side must have a return type which conforms to the declared type of the location specified by the lefthand side. We consider each of the allowed forms for the lefthand side in turn.

- `a:=5`, *ident.* If *ident* refers to a local variable, to a routine or iter argument, or to **`res`** or **`exception`**, then the assignment is directly performed. If the variable has an abstract, reference, or bound type, the variable will be assigned a *reference* to the object returned by the righthand side. If the variable has value type, then the *object* itself will be copied to the variable.

  If *ident* does not refer to one of these variables, then it must refer to a routine in the class in which it appears. In this case, the statement is syntactic sugar for a dotted call on **`self`** with a semantics that is described in the next item.

- `b(7).c:=5`, `A::d:=5` *ident.* These forms are syntactic sugar for calls on a routine named *ident* with the righthand side as an argument. The two examples would be transformed into `b(7).c(5)` and `A::d(5)`.

- `[3]:=5`, `e[7,8]:=5`, `B::[7,8,9]:=5`. These forms are syntactic sugar for calls on a routine named **`aset`** with the array index expressions and the righthand side as arguments. The three examples would be transformed into **`aset(3,5)`**, **`e.aset(7,8,5)`**, and **`B::aset(7,8,9,5)`**.

- `_:=f`. When the lefthand side is an underscore, the righthand side is evaluated and the result is ignored.

- `g:INT:=5`, `h::=5` *ident.* This form declares a new local variable and assigns to it. If a type specifier is not provided, then the declared type of the variable is the return type of the expression on the righthand side.

Tuple assignments are used to extract or modify the components of tuple value objects.

```
#(a,b):=c
#(a:,b):=c
#(a,_):=c
c:=#(a,b)
c:=#(a,_)
#(a,b):=#(b,a)
```

The first form has what looks like a tuple object constructor on the lefthand side. Each component must be the lefthand side of one of the simple assignment forms described in the last section. The righthand side must return a value object with the same number of components as appear on the lefthand side. Each component is assigned as described in the last section. A subset of an object's components may be extracted by using underscores "`_`" for unwanted components on the lefthand side (eg. **`#(a,_):=c`**).

The second form is used to assign to some of the components of a value object. The righthand side looks like a value object constructor but may also contain underscores "`_`". The lefthand side must describe a location of value type with as many components as are specified by the construct on the right. The righthand side expressions are first evaluated from left to right. If the lefthand side is a local variable of value type, then each righthand side expression is assigned to the corresponding object component. Components corresponding to underscore positions are left unchanged.

2.3.3 Conditionals

```
if a>5 then foo elsif a>2 then bar else error end
```

corresponding to underscore positions are left unchanged.`if`-statements are used to conditionally execute statement lists according to the value of boolean expression, as usual in most programming languages.

2.3.4 Loops

```
loop ... end
```

`loop`-statements are used to perform iteration. Their real power arises in conjunction with iters. Iters are like routines except that their names end with "`!`", their arguments may be marked with "`!`" and their bodies may contain the statements "`yield`" and "`quit`".

Storage is associated with each iter call to keep track of its execution state. When a loop is first entered, the execution state of all enclosed iter calls is initialized. The first time each iter call is executed in a loop, the expressions defining `self` and each of the arguments are evaluated left to right. On subsequent calls, however, only the expressions for arguments that are marked with a "`!`" are re-evaluated. `self` and any arguments not marked with a "`!`" retain their earlier values.

When an iter is called, it executes the statments in its body in order. If it executes a `yield` statement, control is returned to the caller and the current value of `res`, if any, is returned. Subsequent calls on the iter resume execution with the statement following this `yield` statement. If an iter executes `quit` or reaches the end of its body, control passes immediately to the end of the enclosing loop in the caller. In this case no value is returned from the iter.

2.3.5 Termination and Yield Statements

```
return
yield
quit
```

`return` statements may only be executed in the body of a routine and cause immediate return from the routine. Routines return the value of `res` if they are declared to have a return value. Iters may not contain `return` statements.

`yield` statements may only be executed in the body of an iter and serve to return control to the point where the iter was called. The iter yields the value of `res` if it is declared to have a return value.

`quit` statements may only be executed in the body of an iter and cause immediate termination of the iter, the enclosing loop, and all fellow iters in the same loop.

2.3.6 Typecase Statements

```
typecase a
   when $A then ...
   when INT, FLT then ...
   else ... end
```

Operations that depend on the runtime type of an object that is held by an abstract variable may be performed using `typecase` statements. *ident* must name a local variable, argument, or return value of a routine or iter. If the `typecase` appears in an iter, then *ident* must not refer to a "`!`" argument because the type of object that such an argument holds could change. *ident* may also be a local variable declared and assigned to in the `typecase` statement itself. This case is syntactic sugar for a statement assigning or declaring the variable followed by the `typecase` statement listing only the variable's name.

On execution, the type of the object held by the variable is checked for conformance with the successive type specifiers in each *type_spec_list*. The statement list following the first type specifier it conforms to is executed. Within that statement list, however, the type of the variable is taken to be the type specified by the conformant type specifier. All type checking within that statement list is done as if the variable were declared by the corresponding type specifier. If the object's type doesn't conform to any of the listed type specifiers, then the statements following the `else` keyword are executed. The declared type of the variable is not changed within the `else` statement list. If the value of the variable is `void`, then it is an error to attempt to execute a `typecase` statement.

2.3.7 Exceptions

```
protect ...
   against $E then ...
   against E1, E2 then ... end
```

`protect`-statements define exception handlers. Execution begins with the statement list following the `protect` keyword. As long as no exception is raised, the statements in this list are executed to completion. If an exception is raised during the execution of these statements, then the object held by the built-in variable `exception:$EXCEPTION` is used to select a handler. Its type is checked for conformance against successive type specifiers in the lists following the `against` keywords. The statement list following the first type specifier to conform is executed. If the type of the object held by `exception` doesn't conform to any of the type specifier lists, then the same exception is raised to the next dynamically enclosing `protect` statement.

```
exception := ...; raise
```

`raise` statements are used to raise exceptions. Programs typically assign an object to the built-in variable `exception:$EXCEPTION` which describes the nature of the exception before executing a `raise` statement.

2.3.8 Expression Statements

```
foo(1,2)
```

A statement may consist of just an expression with no return value executed for its side-effects.

2.4 Expressions

Sather expressions are used to compute values or to cause side-effects. They may have a *return value* in which case they have a *declared return type*. Expressions may appear in statements, in the bodies of routines and iters, as the `pre` and `post` tests for routines and iters, as initializers for constant, shared, and object attributes, and as the specifier and default value for integer class parameters.

2.4.1 Local Access Expressions

```
a
self
```

The simplest expressions return the contents of a routine or iter argument or local variable. Unless otherwise noted, these may only appear in the bodies of routines and iters. There are several cases:

- *ident* may name a local variable of the routine or iter in which it appears. The return type is the declared type of the local variable.
- *ident* may name an argument of the routine or iter in which it appears. These expressions may also appear in `pre` and `post` clauses. The return type is the declared type of the argument.
- `self` refers to the object on which the routine or iter was called. It may also appear in `pre` and `post` clauses. Its type is the same as the class in which it appears.
- `res` may appear in the body or `post` clause of a routine or iter that returns a value. It holds the value that will be passed to the caller when a routine returns or an iter yields. Its type is the declared return type of the routine or iter.
- `arg` may appear in the body and `pre` and `post` clauses of routines and iters. It is defined in routines and iters which have only a single argument whose name is not specified.
- `exception` of type `$EXCEPTION` refers to the current exception object. It is typically accessed within an `against` clause of a `protect` statement.

2.4.2 Routine and Iter Call Expressions

All other expressions consisting of a single identifier are syntactic sugar for calls of routines or iters on `self`.

```
a(5,7)
b.a(5,7)
A::a(5,7)
```

The most common expressions in Sather programs are routine and iter calls. *ident* names the routine or iter being called. The object to which the routine or iter is applied is determined by what precedes *ident*. If nothing precedes it, then the form is syntactic sugar for a call on `self` (eg. `a(5,7)` is short for `self.a(5,7)`). If *ident* is preceded by an expression and a dot "`.`", then the routine or iter is called on the return value of the expression. If *ident* is preceded by a type specifier and a double colon "`::`", then the routine or iter is taken from the interface of the specified type with `self` set to the value of `void` for that type.

Sather supports routine and iter *overloading* which means that the declared argument types and the use of a return value are used to choose between routines or iters with the same name. The *expr_list* portion of a call must supply an expression corresponding to each declared argument of the routine or iter. The return type of these expressions must conform to the declared types of the corresponding arguments.

While overloading is resolved statically by the compiler, Sather also supports *dynamic dispatch* on the type of `self`. This happens when the declared return type of the expression *expr* on which the call is made is abstract. The routine or iter which is actually executed will be from the class corresponding to the runtime type of the returned object. The name of the called routine or iter may be overloaded in the abstract class. In this case, the choice is resolved statically in the abstract class as described above. Each routine or iter in an abstract class uniquely corresponds to a routine or iter in each descendant class which has the same name, argument types, and presence or absence of a return value. It is this corresponding routine or iter which is called at runtime.

Direct calls on a class's routines and iters may be made using the double colon "`::`" syntax. *type_spec* must specify a reference, value, or external class. The name *ident* and the argument number and types and the presence or absence of a return value are used to select a routine or iter from the specified class as described above. The value of `self` in such calls is `void`.

2.4.3 Constructor Expressions

```
#R(a:=1,b:=2,c:=3)
#V(1,"test")
#R
#
#ARR{INT}(a,b,4)
#ARR{INT}(asize:=17)
#LINK(link:=#LINK(link:=#1))
```

Constructor expressions start with "`#`" and are used to build data structures. If *type_spec* appears, it specifies the type of the object returned and must be a reference or value type. If no type is specified, the type is inferred from the use of the expression. If it is assigned to a variable with a declared type, then that type is taken to be the return type of the constructor (it must be either a value or a reference type). If it is assigned to a local variable using the "`::=`" syntax, then the type is a tuple whose component types are the return types of any "*cons_elt*"'s which appear.

In reference object constructors, the *cons_elt* elements specify attribute values and array elements. Attribute specifiers consist of the name of the attribute, "`:=`", and an expression for the value of the attribute. Within the class itself, any attribute may be specified in this way. Within other classes, only those attributes whose "writer" routine is public may be specified. Attributes specifiers may appear in any order and are all optional. Unspecified attributes will be set to the value of the corresponding initializing expression in the class definition. If there is no initializer, then the attribute is set to "`void`". Any specified expressions are evaluated left to right.

Constructors for reference objects which include `ARR` may explicitly give a value to `asize` (eg. `#ARR{INT}(asize:=17)` unless it is declared as a constant in the class. If `asize` is specified, then the elements of the array portion are initialized to "`void`". Alternatively, expressions giving values for the array elements may be listed in order after any attribute specifiers. In this case, the size of the array portion is taken to be the number of expressions provided. If "`asize`" is a constant, then it must equal the number of expressions.

When a constructor component expression consists of a sharp sign followed by digits (eg. `#5`), it represents a textually earlier object in the outermost enclosing constructor. The digits specify how many constructors from the front of the constructor the object is. This notation is used for data structures with cycles.

2.4.4 Bound Routines and Iters

```
#ROUT(2.plus(_))
#ITER(_:INT.upto!(5))
```

Bound routines and iters generalize the "function pointer" and "closure" constructs of other languages. They bind together a reference to a routine or iter and zero or more argument values (possibly including `self`).

The outer part of the expression is `#ROUT(...)` for bound routines and `#ITER(...)` for bound iters. These surround an ordinary routine or iter call in which any of the arguments or `self` may be replaced by the underscore character "`_`". These arguments will be specified when the bound routine or iter is eventually called. Optional `:`*type_spec* clauses are used to specify the types of underscore arguments or the return type and may be necessary to disambiguate overloaded routines or iters. The expressions in this construct are evaluated from left to right and the resulting values are stored as part of the bound routine or iter. If `self` is specified by an underscore without type information, the type is taken to be `SAME`.

Each bound routine defines a routine named `call` and each bound iter defines an iter named `call!`. These have argument and return value types that correspond to the bound type descriptor. Invocation of this feature behaves like a call on the original routine or iter with the arguments specified by a combination of the bound values and those provided to `call` or `call!`. The arguments to `call` or `call!` match the underscores positionally from left to right (eg. `i::=#ROUT(2.plus(_)).call(3)` is equivalent to `i::=2.plus(3)`).

2.4.5 Syntactic Sugar Expressions

```
a+b
x<7
```

Several syntactic constructs in Sather are simply syntactic sugar for corresponding routine calls:

- "*expr1* `+` *expr2*" is transformed to "*expr1*`.plus(`*expr2*`)`".
- "*expr1* `-` *expr2*" is transformed to "*expr1*`.minus(`*expr2*`)`".
- "*expr1* `*` *expr2*" is transformed to "*expr1*`.times(`*expr2*`)`".
- "*expr1* `/` *expr2*" is transformed to "*expr1*`.div(`*expr2*`)`".
- "*expr1* `^` *expr2*" is transformed to "*expr1*`.pow(`*expr2*`)`".
- "*expr1* `%` *expr2*" is transformed to "*expr1*`.mod(`*expr2*`)`".
- "*expr1* `/=` *expr2*" is transformed to "`not (`*expr1*`=`*expr2*`)`".
- "*expr1* `<` *expr2*" is transformed to "*expr1*`.is_lt(`*expr2*`)`".
- "*expr1* `<=` *expr2*" is transformed to "*expr1*`.is_leq(`*expr2*`)`".
- "*expr1* `>` *expr2*" is transformed to "*expr1*`.is_gt(`*expr2*`)`".
- "*expr1* `>=` *expr2*" is transformed to "*expr1*`.is_geq(`*expr2*`)`".
- "`-` *expr*" is transformed to "*expr*`.negate`".
- "`[`*expr_list*`]`" is transformed to "`aget(`*expr_list*`)`".
- "*expr1*`[`*expr_list*`]`" is transformed to "*expr1* `.aget(`*expr_list*`)`".
- "`(`*expr*`)`" is transformed to "*expr*".

2.4.6 Boolean Expressions

```
0<=x and x<10
x=2 or x=3
not s.is_empty
```

`and` expressions are boolean-valued as must be the two component expressions. The first expression is evaluated and if it is `false`, this is immediately returned as the result. Otherwise, the second expression is evaluated and its value is returned.

`or` expressions are boolean-valued as must be the two component expressions. The first expression is evaluated and if it is `true`, this is immediately returned as the result. Otherwise, the second expression is evaluated and its value is returned.

`not` expressions are boolean-valued and as must be the contained expression. The value returned is the logical complement of the expression.

2.4.7 Equality

```
x=5
```

Equality test expressions return boolean values. The two component expressions must each have a return value but are otherwise arbitrary. Regardless of their types, the left side is evaluated first and then the right side. If one of the two sides returns a value type, then the other side must return the same type or the result is automatically `false`. If they both return the same value type, then the return values are compared bitwise for equality. If the two sides return bound or reference objects, then they are compared for reference equality.

2.5 Special Features

This section describes some of the features of classes that are automatically defined or have special properties.

`type`: Every type defines the routine "`type:TYPE`" which may not be redefined. It returns a value object that identifies the type. It is useful in tests such as `a.type=FOO::type`. A string representation of an object's type may be obtained with "`a.type.str`".

`copy`: Each class defines "`copy:SAME`" to return a copy of an object. Because bound objects are immutable, they simply return `self`. By default, value classes also just return `self` but reference classes return a one-level shallow copy. Classes should redefine this to be most appropriate for the structure they represent.

`destroy`: Bound and reference classes have the built-in routine `destroy` which may not be redefined. It is used to aid debugging and to improve the performance of programs. When checking is enabled, it is used to mark an object that shouldn't be referenced again. Any such reference will raise an exception. When checking is not enabled, this call will explicitly free the object and save the garbage collector some work. Sather is garbage collected and there is no need to explicitly deallocate objects.

`while!`, `until!`, **and** `break!`: Each class has three predefined iters which may not be redefined. `while!(BOOL!)` yields when its argument is true and returns when it is false. `until!(BOOL!)` yields when its argument is false and returns when it is true. `break!` immediately quits. These may be used to imitate each of the traditional loop constructs. For example, when the `while!` iter is called at the top of a loop the behavior is like a "`while do`" construct and if it is called at the bottom of the loop the behavior is like a "`do while`" construct. The `break!` iter immediately transfers control to the statement following the innermost loop it is called within, like a traditional `break` statement.

invariant: If a routine with the signature "**invariant:BOOL**", appears in a class, it defines a *class invariant*. When checking is enabled, it is evaluated after each non-private routine of the class returns or each non-private iter yields. It is also evaluated when objects of this type are constructed. If the routine returns false, an exception is raised.

main: Execution of a program begins at the routine named **main** in a class specified to the compiler. **Main** may be declared to have an argument of type `ARRAY{STR}` which will contain any command line arguments in the order specified when the program is called. It may be declared to have a single return value of type `INT` which becomes the exit code of the program when it finishes execution.

2.6 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 only a short description of these classes. The detailed semantics and precise interface is specified in the Sather class library documentation.

- `$OB` is automatically an ancestor of each abstract and reference class. It may be used to declare variables that can hold any reference object. It supports `type:TYPE`, `id:INT`, `copy:SAME`, `destroy`, `str:STR`.
- `BITS` may be inherited by value classes which represent a single field of data. The descendant may define the two constants `bsize:INT` and `balign:INT` to specify the size in bits of the object and its alignment requirements. The default value of `void` is all zeros.
- `BOOL` defines value objects which represent boolean values. The value of `void` is `false`.
- `CHAR` defines value objects which represent characters. The value of void is `'\0'`.
- `STR` defines reference objects which represent strings.
- `INT` defines value objects which represent machine-dependent integers. The size is implementation dependent and must be large enough to hold a machine pointer value. The two's complement representation is used to represent negative values. Bit operations are supported in addition to numerical operations. The value of `void` is 0.
- `INTINF` defines reference objects which represent infinite precision integers. They support arithmetic operations but do not support bit operations.
- `FLT`, `FLTD`, `FLTE`, and `FLTDE` define value objects which represent floating point values according to the single, double, extended, and double extended representations defined by the IEEE-754-1985 standard. The value of `void` is `0.0`.
- `ARR{T}` is a reference class defining dynamically-sized arrays of elements of type `T`. Array indices start at zero. Classes which inherit from this are called *array classes*. They allocate space for the array and the attribute `asize:INT` whose value is the number of elements in the array. They also define a number of operations including `copy(INT):SAME`, `aget(i:INT):T`, `aset(i:INT, T)`.
- `$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.

- SYS defines a number of routines including deep_str(ob:$OB):STR, deep_copy(ob:$OB):$OB and deep_equal(ob:$OB):BOOL for manipulating arbitrary data structures.
- TYPE defines the value objects returned by the type routine. "str" applied to these objects returns a string with the name of the class.
- $REHASH, which defines the single routine rehash, should be inherited by any class whose objects need to perform special operations when they are moved or copied. rehash is called on such objects after the construction of literal structures, after deep copy operations, after the construction of deep data structures from a string or file description, and after any object movement during garbage collection.
- $EXCEPTION is inherited by all exception types.

After this concise introduction into the base language Sather we will show how to extend Sather for parallel computation in the remainder of this report.

3 pSather on a Shared Memory Machine

In a first step we extend Sather for a machine model where multiple processors have equal access to the same shared memory. Examples for such architectures include DASH [47], Sequent [50], and Sun SPARCcenter 2000 [25], SPARCstation 10/xx etc. In order to make use of multiple processors the language must provide means to create and synchronize multiple parallel *threads* of instructions. This version has been running on the Sequent and SPARCstation since early 1991.

We will not give many examples and motivations for design decisions in this section because this part of pSather has not changed much since [27].

3.1 Non-Blocking Calls: Creating Threads

Each regular pSather routine call without return values can be made to a simple *non-blocking* (or *asynchronous*) call by preceding it with a *deferred assignment operator* (`:-`). It will become clear in section 3.3.4 why we call the `:-` operator deferred assignment instead of thread forking operator (although forking a new thread is part of the semantics of the `:-` operator) and how we deal with routines with return values.

The `:-` operator extends the Sather expression statement syntax[1] as follows:

expr_stmt ⇒ *[*`:-`*] expr*

We will call the asynchronous streams of instructions initiated by deferred assignments *threads*. A non-blocking call creates a new thread and installs it as a *child thread* to a running thread. Parent threads may terminate earlier than their children. A running pSather program has a collection of threads. The whole program terminates when all threads created during the execution of the program have terminated. Except for the total amount of memory in the system there is no limit on the number of threads in a program. Scheduling of threads is *fair* in the sense that every ready (non-blocked) thread will eventually run.

Sather exception handling (cf. 2.3.7) has some subtle interactions with the parallel constructs of pSather: Each thread starts a new chain of exception handlers directly attached to the system exception handler. This implies that exceptions that are not caught within the thread are passed directly to the system exception handler and not to the parent thread. This semantics is necessary because child threads may in general live longer than their parents.

The simplest way to synchronize threads is the `cobegin`-statement. The creation of several threads by deferred assignment can be enclosed in a `cobegin`-statement that has the following syntax:

cobegin_stmt ⇒ `cobegin` *stmt_list* `end`

A thread executing a `cobegin`-statement will not continue with the statements following the `cobegin`-statement until all child threads that were created during the execution of the body of the `cobegin`-statement are terminated. Note that this includes all threads that are created by routines called in the body. If no threads are created in the body, the `cobegin`-statement has no effect.

The `cobegin`-statement belongs to a group of structured statements which have to perform a termination action. In the case of the `cobegin`-statement, correct termination means waiting for all child threads to terminate. This leads to the following semantic rules:

- If an exception is passed on beyond the end[2] of a `cobegin`-statement, all threads are first synchronized. One may look at the `cobegin`-statement as an implicit exception handler; it

[1] The pSather syntax is an extension of the syntax of Sather in [61]

[2] We say that an exception is passed on beyond the end of a statement if it is raised (explicitly by the `raise`-statement, or implicitly by an inner exception handler that could not catch the exception or the runtime system) within the body of that statement and cannot be handled in the body.

handles all exceptions by first properly synchronizing all threads and then passing the exception on to the next outer handler.

- `return` or `quit` from within a `cobegin`-statement does all the synchronization before leaving the routine or the iter, respectively.
- `yield` from within a `cobegin`-statement in an iterator does not synchronize the threads as long as control returns to the iterator. If however the loop containing the iterator invocation is terminated, the `cobegin`-statement synchronizes the threads.
- Iter calls from within a `cobegin`-statement which belongs to a loop enclosing the `cobegin`-statement do not affect synchronization immediately. However, if one of these iter calls terminates the loop, all threads are synchronized before termination of the loop.

3.2 Example: Fractals I

The following example is a classical parallel algorithm. We compute the Mandelbrot set for a given 2-dimensional range into a pixel array. For each pixel we fork `man_pixel` as a separate thread. For many machines this might be too fine grained, but it shows one possible use of the `:-` operator.

```
class FRACTALS is
   attr left, right, bottom, top, hstep, vstep: FLT;

   private man_pixel(p:ARRAY2{BOOL}): {INT,INT} is
      #(i:, j:):=indices;
      xpos::=i.flt*hstep+left; ypos::=j.flt*vstep+bottom;
      x::=xpos; y::=ypos;
      loop
         100.times; -- Iterator
         x2::=x*x; y2::=y*y;
         if (x2+y2>=4.0) then p[i,j]:=true; return; end;
         x:=(x2-y2)+xpos; y:=x*y*2.0;
      end;
      p[i,j]:=false;
   end;

    mandel(p:ARRAY2{BOOL}) is
      -- Display part of the mandelbrot set in a 2-d black/white
      -- pixel array
      hstep::=(right-left)/p.asize1.flt;
      vstep::=(top-bottom)/p.asize2.flt;
      cobegin
         -- Compute each pixel in parallel.
         loop
            -- ''inds_tup2'' is an iterator producing all index tuples
            -- for a 2-dimensional array.
            :- man_pixel(p, p.inds_tup2);
         end;
      end;
   end;
end;
```

We will see more synchronization constructs in the next section.

3.3 Synchronization: Gates

The central synchronization construct is based on two concrete classes of *gates* with a set of predefined attributes and routines. Gate is a new name for the "monitor" in [27] because the name "monitor" is already used in computer science for similar but not identical concepts leading to unnecessary confusion. Although pSather gates add some new functionality they are similar to the classical shared memory synchronization primitives (semaphores [22], monitors [38] [33] etc.). Not surprisingly, one can easily model all the classical synchronization primitives using gates.

There are two versions of gates sharing the same basic functionalities: the parameterless **GATE0** class and the parameterized **GATE{T}** class. The difference is that objects of type **GATE{T}** contain a value of type T whereas objects of type **GATE0** contain no value. Both classes are predefined and can be used with the functionalities described below.

These gate classes do not alter the syntax or semantics of Sather's existing type system, and can be used like other reference classes. Each class provides both interface and implementation for its predefined routines. User classes may include either **GATE0** or **GATE{T}** (to inherit the implementation according to the Sather rules). The only thing special about including a gate class is that the predefined operations may not be redefined by the inheriting class. The reasons for this are that on one hand these atomic operations need special support from the compiler and the runtime-system and on the other hand we would like to protect the user from redefining the synchronization primitives. The abstract view of a gate of type **GATE{T}** is that it contains the following unnamed attributes:

- a queue of values whose types must all conform to T (the type parameter of **GATE{T}** class),
- a set of associated threads,
- a lock status (set to locked or unlocked) and
- a binding status (set to bound or unbound).

These unnamed attributes are not directly modifiable by the programmer. There is, however, a set of predefined routines that allows the programmer to act on these attributes, as we will see below.

3.3.1 Controlling the Locked/Unlocked Status

The `lock`-statement and the `try`-statement can be used to *lock* a gate. We will refer to these statements as locking statements. The syntax of the `lock`-statement is:

lock_stmt ⇒ **`lock`** *expr_list* **`then`** *stmt_list* **`end`**

The expression list in the header of the `lock`-statement consists of *gate expressions*. A gate expression specifies a gate and can be followed by a predicate as described in section 3.3.2. If all of the denoted gates are in the unlocked status or locked by the executing thread, then they are all atomically locked and the statement list in the body of the `lock`-statement is executed. If any of the gates is not available for locking, the executing thread is suspended. The executing thread is woken up again as soon as all the gates become available for locking. After the execution of the body statements, the lock status existing before the execution of the locking statement is restored. That is, if a gate was already locked by the executing thread, it is still locked after the lock-statement is executed, otherwise it is unlocked.

The `lock`-statement, like the `cobegin`-statement, belongs to the group of structured statements which have to perform a termination action. In the case of the `lock`-statement proper structure termination means restoring the proper lock status for all the gates locked by the statement. This leads to the following semantics rules:

- If an exception is passed on beyond the end of a `lock`-statement, the corresponding gates are first unlocked. One may look at the `lock`-statement as an implicit exception handler that handles all exceptions by properly unlocking all gates and passing on the exception to the next outer handler.

- `return` or `quit` from within a `lock`-statement does all the unlocking before leaving the routine or iter, respectively.

- `yield` from within a `lock`-statement does not change the lock status of the locked gates implicitly as long as control returns to the iterator. But if another iterator terminates the whole loop such that after yielding, control never returns to the same iterator invocation again, open `lock`-statements in the iterator are properly terminated. This implies that an iterator may keep a lock on a gate during the execution of a loop.

- Iter calls from within a `lock`-statement which belong to a loop enclosing the `lock`-statement do not affect the lock status immediately. However, if one of these iter calls terminates the loop, the proper lock status is restored.

The `try`-statement is a non-blocking variant of the `lock`-statement. The syntax of the try-statement is:

try_stmt ⇒ `try` *expr_list* `then` *stmt_list* [`else` *stmt_list*] `end`

The expression list again consists of gate expressions. If all of the gates specified in the header are in the unlocked status or are locked by the executing thread, then the `try`-statement has the same semantics as the `lock`-statement. If any of the gates is not available for locking, the executing thread is not suspended. If the optional statement list is given after `else`, it is executed. If no `else`-branch is specified, execution continues after the `try`-statement is executed. All the special rules concerning abnormal termination of the `lock`-statement apply to the `then`-part of the `try`-statement.

Early restoration of the original locking status of a gate can be done by using the `unlock`-statement which has the syntax:

unlock_stmt ⇒ `unlock` *expr*

An `unlock`-statement can only appear inside the scope of a `lock`-statement or the `then`-branch of a `try`-statement. Only a gate that is in the gate list of a surrounding locking statement can be specified in the gate expression of the `unlock`-statement. Because of possible aliasing among reference objects this condition must be checked dynamically by the runtime system. A gate can only be unlocked once per locking statement, either by an `unlock`-statement or when the locking statement terminates as described above. In order to guarantee this property of gates statically, the compiler checks that `unlock`-statements do not occur in loops enclosed by `lock`-statements or `then`-branch of `try`-statements.

Figure 2 shows the possible state transitions [3] of a gate between locked and unlocked status. A locked gate may only be locked again by the same thread.

3.3.2 Predicates in Gate Expressions

The following set of built-in predicates can be used to test the various status of a gate object.

```
g.is_bound
g.is_unbound
g.has_threads
g.no_threads
```

[3] Note that labels attached to the state transitions consist of two parts separated by a vertical bar where the first part defines the name of the event leading to the state transition and the second (optional) part defines some action that is executed atomically together with the state transition.

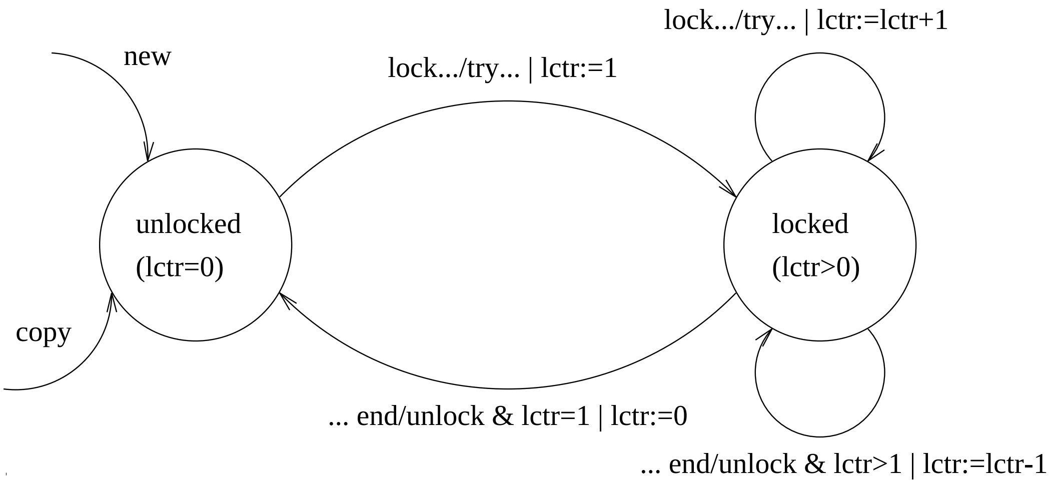

Figure 2: Possible state transitions of the gate lock status

The predicate `g.is_bound` returns true, if `g` is in the bound status. The predicate `g.is_unbound` returns true, if `g` is in the unbound status (section 3.3.3 has more details on the binding status).

A gate `g` can be used to control the execution of a thread as described in section 3.3.4. The created threads are *attached* to `g`. If `g` is a of type `GATE{T}`, the values returned by the threads after completion are stored in `g`. The boolean predicates `has_threads` and `no_threads` test whether any threads are attached to `g`.

The gate expressions in the header of a locking statement can be specified with one of the built-in predicates. The semantics is that a gate is successfully locked if it is available for locking and also in a status in which the predicate is true[4]. If the gate status does not satisfy this condition, in the case of a `lock`-statement the executing thread is suspended until the condition becomes true, after which it is normally locked. In the case of a `try`-statement the alternative sequence of statements (if any) is executed.

3.3.3 Controlling the Binding Status

The *binding status* of a gate of type `GATE{T}` is associated with a *queue of values* of type `T` that is stored in the gate. The gate is in the bound status if the value queue is not empty. Otherwise, the gate is in the unbound status.

The binding status of a gate of type `GATE0` is associated with a binding counter instead of a value queue[5].

There are several built-in operations on gates that manipulate the value of a gate or the queue of values. A thread that executes one of these operations on a gate locked by another thread, is suspended until the gate is unlocked. The following operations are provided to store/remove data items of type `T` in/from a gate `g` with type `GATE{T}`:

```
g.set(<expr>)
g.enqueue(<expr>)
x:T:=g.take
```

[4] This is the reason why we have both positive and negative status predicates.

[5] The semantics of a `GATE0` object are actually very similar to those of counting semaphores, where the `take`- and `read`- operations correspond to the P- und V-operations on semaphores, respectively.

```
x:T:=g.read
g.clear
g1:GATE{T}:=g.copy
```

All the operations explained below work as well for gates of type `GATE0`. The only difference is that they neither take an argument to be bound to the gate nor return a value of the gate. They just keep track of the number of binding operations.

`set`-operation: The argument of type `T` is stored as a new first element in the value queue of `g` and overwrites a previous value if `g` was already bound. `g` is in the bound status after the operation. The other elements in the value queue are not affected. This is the *destructive write* operation on a gate.

`enqueue`-operation: The `enqueue`-operation enqueues its argument to the tail of `g`'s value queue. If the gate was unbound (i.e. the queue was empty) before the operation, the binding status of the queue changes to bound. When compared to `set` this is the *non-destructive write* operation on gates. It is a bad programming practice to mix destructive and non-destructive write operations on the same gate in the same part of a program.

`take`-operation: If `g` is unbound when the operation is executed, then the executing thread is suspended until the gate becomes bound. If `g` is of type `GATE{T}` and is bound when the operation is executed, then the operation returns the first element stored in the value queue `g` and removes it from the queue. If the queue of values attached to `g` is empty after execution, the take-operation puts `g` into the unbound status. `take` is the *destructive read* operation on gates.

`read`-operation: If `g` is unbound when the operation is executed, then the executing thread is suspended until the gate becomes bound. If `g` is of type `GATE{T}` and is bound when the operation is executed, then the operation returns the first element in the value queue of `g` without removing it from the queue. `g` remains bound. If `g` is of type `GATE0` and is bound when the operation is executed, then the read-operation has no effect. According to our classification scheme, `read` is the *non-destructive read* operation on gates.

`clear`-operation: The gate `g` is put in the unbound status and all threads attached to `g` are detached. The suspended threads that are waiting for `g` are not affected by the operation. If `g` is of type `GATE{T}`, the queue of values of `g` is emptied. With the predicate `CONFIG::clear_request` a thread attached to a gate can actively find out that its gate was cleared and, therefore, the thread detached (see section 3.3.4 for more).

`copy`-operation: Like any other object in pSather, gates can be copied. The `copy`-operation returns a new gate object. It the queue of the source object was empty the queue in the new object is empty, too. If the queue of the source gate was not empty the new object gets a queue containing only the first element of the queue in the source object. Copied gates are always unlocked, and bound or unbound depending on whether the queue is empty or not. I.e. `read`-operations on copied gates have the same results as on the original, whereas `take`-operations do not produce the same sequence of results. In our classification `copy` of a non-empty queue consists of a non-destructive read from the source gate and a write into the destination gate. The list of threads attached to `g` (i.e. threads that are controlled by `g` and that are not suspended) are not copied into the new gate. The list of threads suspended on `g` is also not copied.

If there is more then one `take`- or `read`-operation suspended when `g` becomes bound, they are resumed and executed in FIFO order, as far as the gate status allows it. E.g. if two take-operations are suspended, only the first one will be resumed. Similarly, suppose that three take-operations are suspended on a locked gate, and the gate is finally released. If the gate is bound and two values are present in the queue, only the first two of the three take-operations will be resumed one after another (leaving the gate unbound).

Figure 3 shows the possible gate state transitions between bound and unbound. In the state diagram,“bctr” refers to either the binding counter of `GATE0` or the number of queue elements of `GATE{T}`. Again we use the convention that labels belonging to state transitions consist of an event name and an action to be executed atomically, separated by a vertical bar. Note that the request of a state transition not defined in Figure 3 (e.g. `read` from an unbound gate) causes the executing thread to suspend until the gate is in a state from which the transition is defined. Binding operations may only be performed either on an unlocked gate or by the locking thread on a locked gate.

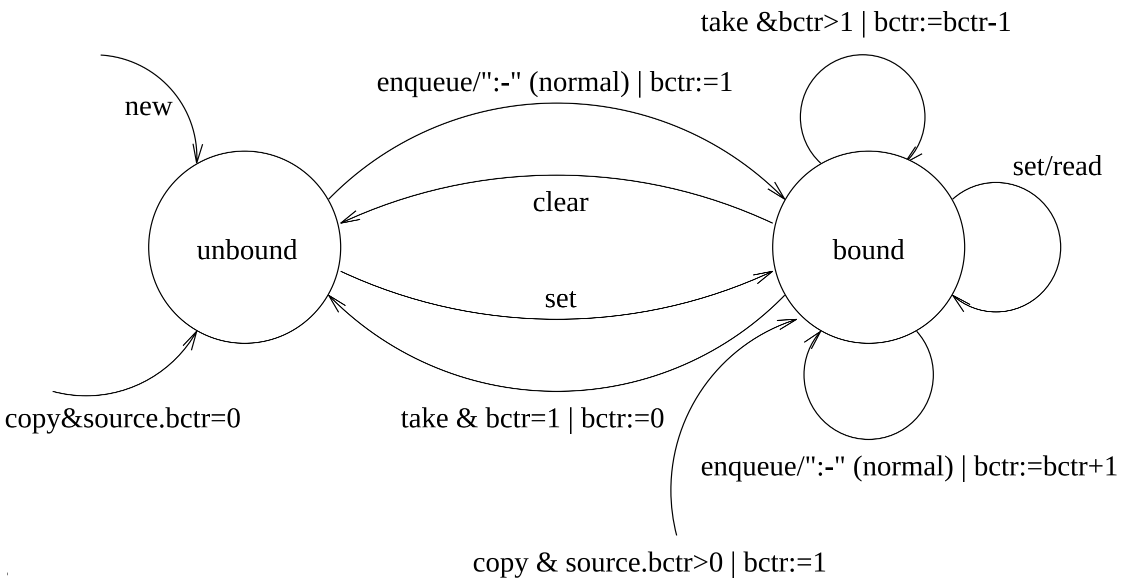

Figure 3: Possible state transitions of the gate binding status

3.3.4 Deferred Assignment

Gates can appear on the left hand side of non-blocking routine calls returning a result[6]. We call this a *deferred assignment* extending the Sather simple assignment statement syntax as follows:

simple_assign_stmt ⇒ *lhs_elt [*`:-` | `:=`*] expr*

The left hand side of the deferred assignment evaluates to a gate `g`. The expression on the right hand side can be a call of any ordinary routine or of a bound routine. If the routine returns a value, then the gate must be of type `GATE{T}` where the return type of the routine must conform to the type parameter of the gate. Non-blocking calls to routines without any return value may be synchronized by a deferred assignment to a gate of type `GATE0`.

[6]It is tempting to allow non-blocking calls to iters [58], as well. For iters that only produce a stream of values but do not take a stream of values (only `once`-parameters) this is possible. It gets more complex for general iters which take and return values at each `yield`-statement, enforcing tightly synchronized execution of the iters and the loop body. But, even in the first case, where it seems possible to let the iter run ahead of the loop body, we have no longer the well behaved coroutine semantics because the iter and the loop body may access common state in shared variables.

If g is currently locked by another thread, the executing thread is suspended until the gate becomes unlocked. If g is not locked by another thread, then a new thread is created that runs asynchronously from the creating thread. If g is of type GATE{T}, then the value returned by the routine in the thread is enqueued after the forked thread has terminated. If the function has no return type, then an enqueue-operation without parameter is performed on the corresponding gate of type GATE0. The execution of the deferred assignment does not lock g. A thread started by a deferred assignment is said to be *attached* to the gate. The predicates has_threads and no_threads indicate whether a gate has attached threads which have not terminated.

The clear-operation detaches all threads attached to a gate. From within a thread the program can check the predicate CONFIG::clear_request:BOOL in order to find out whether its gate has been cleared and it has been detached. A typical application of this is to start multiple threads in parallel for searching a data structure. As soon as the gate gets bound (i.e. one of the results has been found), the result is read from the gate and the gate is cleared. In order to not continue searching after a result has been found, the searching threads can occasionally check whether the gate got cleared, and terminate immediately if this is the case.

One typical application for deferred assignments are simple *futures* [63]:

```
...
g::=#GATE{T};
g :- f(x);
... -- Do something in parallel.
res:=g.take; -- Wait and assign result.
```

3.4 Readers/Writer Synchronization with Gates

In order to demonstrate some of the capabilities of gates in an object-oriented language we show the example of a library class for readers/writer synchronization (Figure 4). The readers/writers problem [19] is the classical problem of synchronizing a number of processes accessing a common resource. The resource accepts only n ($n \geq 1$) readers or 1 writer at the same time. Each thread must start a transaction and identify itself as either a reader or a writer by calling start_read or start_write, respectively. At the end of the transaction the resource is freed by a call to either end_read or end_write. It is a nice example to demonstrate the binding operations on gates. More examples exposing the full flexibility of gates (under the old name "monitor") can be found in [27].

Internally the synchronization is organized around two gates, one for the readers and one for the writers. The binding counter of the readers (writers) gate is equal to the number of readers (writers) accessing the resource in parallel.

start_read waits until the writer gate is unbound and then lets the thread access the resource. end_read adjusts the count of readers via a take-operation on the readers gate.

Synchronization for writers works analogously, except that start_write waits for both the readers and the writer gate to be unbound to proceed. Note that the outer lock-statements both lock the writer first. This ensures fairness between writers and readers, because locking requests are accepted FIFO. When the writer gate is locked, start_writer waits for all readers to terminate, and then sets the writer gate to bound, denying access to any other reader or writer.

3.5 Atomicity and Consistency of Memory Operations

3.5.1 Atomicity

Any programming language must specify a semantics that allows users to program with confidence in what their code will do. The pSather shared address-space model requires specification of the atomicity and consistency of memory operations. As always, this involves a trade-off between clean

```
class RW_SYNC is
   -- Class may be inherited by any other class to provide
   -- a fair readers/writer protocol if accessed by multiple
   -- threads. It is the responsibility of the class designer
   -- to protect his/her routines with the corresponding
   -- "start_x"/"end_x" pairs.

   private attr readers::=#GATE0;
   private attr writer::=#GATE0;

   invariant:BOOL is
      res:= not (writer.is_bound and readers.is_bound)
   end;

   start_read is
      lock writer.is_unbound then
         readers.enqueue;
      end;
   end;

   end_read is
      readers.take;
   end;

   start_writer is
      -- 1. establish place in queue waiting to lock writer among
      -- other readers and writers
      lock writer.is_unbound then
         -- 2. Wait for all readers to terminate
         lock readers.is_unbound then
            -- 3. Set reader guaranteeing exclusive access
    writer.set;
         end;
      end;
   end;

   end_write is
      writer.take;
   end;

end;
```

Figure 4: Class for readers/writers synchronization

language specification and efficient realization based on possible implementations on modern multiprocessors. The basic atomicity rule is: every read or write of a variable of a built-in Sather type or of any reference type is atomic.

This means that during a write operation (assignment) the executing thread has exclusive read and write access rights to the affected part of the memory. Assignments are atomic in the sense that threads cannot be suspended during an assignment. The following example illustrates the problem:

```
class FOO{T} is
   attr found: T;

   search is
      ...
      found := ...
      ...
   end;

   global_search:T is
      cobegin
         ... :- search;
      end;
      res:=found;
   end;
end;
```

Assuming that a programmer writes the above code to do a parallel global search on some data structure. He/she is just interested in the one result that fulfills the requirements for the search. Thus, whatever is written into `found` should be correct. Under the above stated rule for atomicity of memory operations in pSather the program works as expected as long as `T` is one of the built-in value types or a descendant of `$OB`. It, however, may produce wrong results if `T` is a compound value type (tuple). In this case, the user should employ the GATE mechanism (Section 3.3), for example by making `found` be of type `GATE{T}`.

For most architectures and data representations, the atomicity condition is ensured by the hardware design. A primitive data type is usually a single word and is read/written by an indivisible machine operation. But pSather includes built-in types such as `FLTE` (extended float) that may well require multiple machine operations to read or write. Since thread pre-emption is allowed, it would be possible for only part of a `FLTE` variable to be updated in the absence of our atomicity requirement. Even more important, pSather far pointers (Section 4.5) might well require multiple machine operations. The atomicity rule ensures that pointer operations will not be interrupted.

The atomicity rule does not impose any extra burden on the programmer — it has always been assumed implicitly. It does require the programmer to protect with a gate (Section 3.3) any operation on a variable of a basic type that is not built-in and is shared among more than one thread. Since basic type objects can be of arbitrary size, it seems unreasonable to force the implementation to ensure atomicity. The implementation ensures atomicity for built-in types. This does add some constraints to code generation, for example `FLTE` variables might need to be contained to a single cache line and far pointer operations in some architectures could be expensive. But our understanding of current and planned architectures suggests that the extra costs will be minimal. For one thing, pSather local variables and routine parameters are visible only to a single thread and do not require locking. Object and shared attributes are potentially visible to multiple threads but compiler flow analysis can sometimes show that these also need not be locked.

In addition, all gate operations (section 3.3) are atomic. Gates are, among other things, the basic synchronization primitives of pSather and are necessarily atomic. Some gate operations, particularly

```
[Thread T1]                          [Thread T2]

-- "flag" has value false.           if (flag = true) then
x := 3;                                 -- "x" must be 3
g.set;                               end;
flag := true;
```

Figure 5: An example of consistency-ensuring operation.

in distributed systems, can be expensive but these do not seem to be needed very often. Section 9.4 briefly discusses some ways to improve the efficiency of gates.

3.5.2 Memory Consistency

PSather allows threads to freely read and write variables on remote clusters. Multiple threads on different clusters may write to the same variable simultaneously. The atomicity rule guarantees that the variable will have a value written by one of the threads. Another independent question concerns when various threads reading a variable will see the newly written value. This problem arises in modern cache-based processors and is called the memory consistency problem (for which a concise introduction is given in [37]). If every processor cache in a shared-memory machine needs to be synchronized on each write instruction, performance can be greatly reduced[7]. Various levels of consistency among processors have been defined and studied [32]. The strongest of these is sequential consistency [41] which guarantees that every read operation sees the most recent write to the same location.

One condition of sequential consistency is that within a process(or), memory accesses respect the *program order*, ie. the access order specified by the control and data dependences in the program code for that particular process(or) when no reordering takes place. In fact, this condition is also found in other consistency models eg. processor consistency and weak consistency.

Since this is essential for reasoning about the semantics of sequential programs, the semantics of pSather adopts this requirement. The intra-thread consistency requirement specifies that accesses issued from a single thread always obey the program order (based on the code executed by the thread). This does place some constraints on an implementation as we will discuss at the end of this section.

However if we require the program order of a thread to be similarly observed by all other threads, we may not be able to take advantage of the parallelism available in the multiprocessor memory system. For example, if a thread T1 executes:

```
x := 3;
y := 4;
f(x, y);
```

and, `x` and `y` are located on distinct physical memory modules, the writes may happen in parallel without violating the intra-thread order. But the write to `y` may complete before `x`, so that another thread sees the old value of `x` and new value of `y`. In order for this update order to be observed globally, T1 must make sure that the write to `x` is definitely completed before issuing the write to `y`.

We therefore relax the consistency requirements across threads as follows. The update order in thread T1 is observable by other threads when T1 performs a *consistency-ensuring* operation. The following pSather operations are consistency-ensuring: the forking of a child thread, the termination

[7] [31] gives performance measurements for several levels of relaxed consistency.

of a thread, any gate operation, and the termination of a `dist`-statement. The inter-thread consistency rule states that all writes executed by T1 before an ensuring operation will be seen by other threads, before the ensuring operations executes. For example:

- Consider Figure 5. If thread T1 executes an operation on a gate `g1`, all the updates performed by T1 before its access of `g1` are observable by T2. We note that T2 does not need to perform a consistency-ensuring operation.

- A child thread will see the updates done by its parent before its creation.

- Suppose a parent thread executes:

```
cobegin
   ...
   :- f;
   ...
end;
```

  After the `cobegin-end` statement, the parent thread is guaranteed to see the updates done by the child thread before its termination. (The crucial operation here is the child thread's termination; the updates are observable by other threads which may not be its parent.)

The other consistency conditions are that all processors in the machine will eventually see any update of memory, and that all shared attribute broadcasts (cf. `bcast_x` in section 4.5.4) are guaranteed to complete before the next operation is executed.

The relaxed consistency model presented above is a variant of the weak consistency model [23]. However, we need to define consistency on the language level instead of the machine level, as is the case for the traditional consistency definitions.

Both the intra-thread and inter-thread consistency rules place constraints on pSather implementations. The serial intra-thread rule is just the conventional requirement so long as execution is confined to a single processor. Even this entails restrictions on remote operations for systems (such as CM-5) that do not preserve message order. In addition, pSather allows a thread to continue execution on a different processor by invoking a remote procedure. There is also the possibility in some implementations that a thread can be interrupted and later resumed on a different processor. The serial intra-thread rule specifies that pre-emption and the starting or ending of a subthread must include establishing memory consistency between the old and new processors executing the thread.

The weaker inter-thread rule imposes similar implementation requirements. The forking or termination of a child thread and any gate operation forces the completion of outstanding write operations.

We believe that our consistency model, while retaining a simple semantics, will enable future implementations to take advantage of efficient memory mechanisms in large-scale distributed-memory multiprocessors[8]. Note that the weak consistency model is also satisfied by sequentially consistent implementations[9].

[8] In fact, most modern shared memory multiprocessors have weakly consistent memories and provide special instructions to enforce consistency in the memory system.

[9] Our prototype implementation on CM-5 (cf. [48]) actually retains a sequentially consistent model because it is simpler to implement. The usefulness of this consistency model, therefore, remains to be verified.

4 pSather on Distributed Memory Machines

4.1 Machine Model

Providing a shared memory on today's multiprocessors requires maintaining coherence among the caches of the single processors. There exist cache coherence protocols that solve this problem for a moderate number of processors. Massively parallel machines with shared memory do not appear practical with the current technology. In order to overcome these limitations pSather provides a *two-level shared address space*.

Note that a shared address space does not imply having a shared memory. Shared address spaces and shared memories differ with respect to memory latency and bandwidth. *Shared memories*, on the one hand, try to maintain uniform short latencies and high access bandwidth over the whole address space. In modern processors with multi-level memory hierarchies latencies are not really uniformly short, but it is the goal of any memory system to keep up this illusion statistically by good cache-hit ratios. In a *shared address space*, on the other hand, memory latency and bandwidth may vary depending on the address. They are not distributed uniformly over the address space. However, we retain the important property of a *single name space* for a program.

An alternative way of looking at clusters is to partition the set of processors into equivalence sets as follows. We first define the *address set* of a processor p, address-set(p) such that an address $x \in$ address-set(p) iff access latency for x approaches optimal hardware limit when p makes a sufficient number of accesses to x. We assume that for any two processors p, q, their address sets are either equal or disjoint. This is justified from our observations of current and foreseeable machine configurations. Two processors p, q are in the same cluster iff address-set(p) = address-set(q).

It is often important to *replicate* (i.e. make multiple copies of an object in the shared address space such that it is near on more than one cluster) or *distribute* data (i.e. allocate a data structure in the shared address space such that different parts of it appear near on different clusters) in order to achieve the necessary access bandwidth and locality for effective parallel computation.

The model presented in section 3 corresponds exactly to one cluster in the *clustered shared address space model*, presented in this section. Thus the clustered model is a further generalization of the shared memory pSather model by combining multiple shared memories together into a shared address space. We will see that all the new features of pSather introduced in this and the following sections follow from this extension. Figure 6 gives a user view of the model.

This model covers a wide range of parallel systems from shared memory multiprocessors (one single cluster) to distributed memory multiprocessors (each cluster has one processor). pSather has indeed been implemented on both shared memory multiprocessors (e.g. Sequent [50]) and distributed memory multiprocessors (e.g. CM-5 [18]).

4.2 Identification of Clusters

Clusters are identified by numbers of type `INT` in the range between 0 and the number of clusters in the system minus one. Consequently, the `@`-operator (section 4.3) expects operands of type `INT`. Remote calls to non-existing clusters lead to runtime errors. Often we want to deal with whole sets of clusters representing a part of the whole machine. For this purpose we introduce the class `CLUSTER_SET` as a subclass of `FAST_BIT_VEC` in the Sather library which provides bit sets and all the usual set operations. `CLUSTER_SET` adds a `clusters` iterator which yields the clusters in the set in

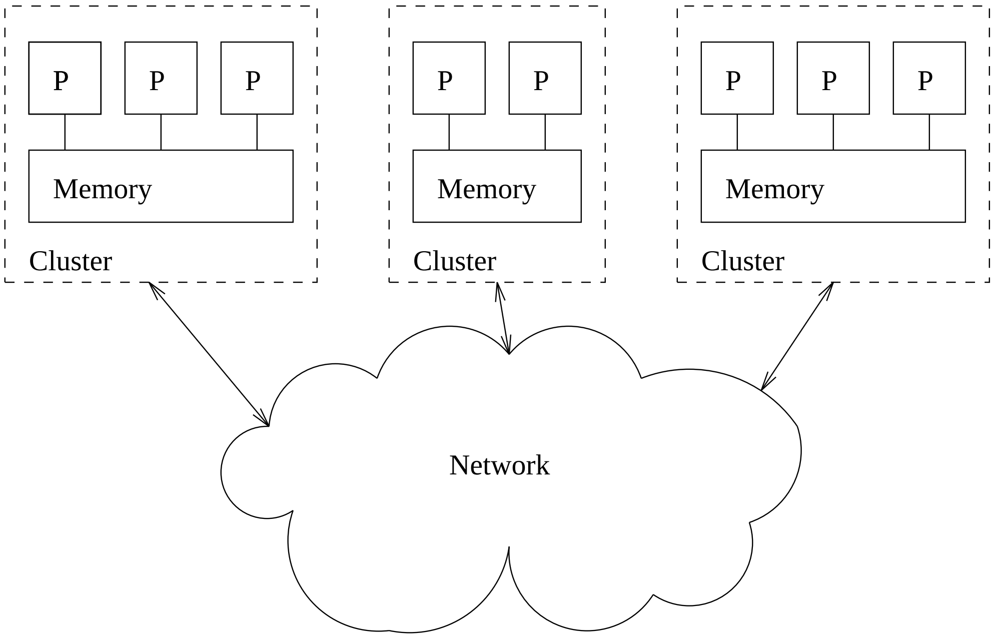

Figure 6: Clustered machine model in pSather.

ascending order:

```
class CLUSTER_SET is
   -- bit-set [0..CONFIG::nr_of_clusters-1]
   include FAST_BIT_VEC;
   ...
   iter clusters!: INT is ... end;
   ...
end;
```

The class `CONFIG` provides some standard information about the system:

```
class CONFIG is
   ...
   constant num_clusters: INT;
   shared all_clusters: CLUSTER_SET;
   ...
   iter clusters!:INT is ... end;
   current_cluster: INT is ... end;
   working_set(n:INT):CLUSTER_SET is ... end;
   ...
end;
```

- `num_clusters` returns the number of clusters in the current configuration.
- `all_clusters` is a shared attribute with a constant cluster set of all clusters in the machine.
- A `clusters` iterator yields the integer identifier of all clusters available in the current configuration. We might use it as follows:

```
loop
  :- worker @ clusters!;
end;
```

- A query function `current_cluster` returns the identifier of the cluster on which the current thread is executing.
- `working_set` returns a set of `n` clusters according to some internal load balancing criteria. The general idea is that these are the n clusters that are expected to be the least active in the near future.

4.3 Remote Routine Calls

Since accesses to far data cost considerably more than near memory accesses it is often important to move the execution to the data. For this purpose pSather provides the `@`-operator with the following syntax:

call_expr ⇒ *[expr* `.` | *type_spec* `::`*] ident [*`(` *expr_list* `)`*] [*`@` *expr]*

The expression after the `@`-operator designates the cluster on which the corresponding procedure call will be executed. Note that the `@`-operator works both with blocking and non-blocking calls.

A thread consists of a stack of *subthreads* on different clusters. Only the top subthread on this stack can be active at any time. In the case of a *remote blocking call* the active subthread is suspended and a new subthread is set up on the cluster designated after the `@`-operator and pushed on the subthread stack. The new subthread is inserted into the ready queue of the scheduler on the appropriate cluster. When the top subthread terminates it is popped from the stack and the new top subthread eventually resumes execution.

Another view of this model is that the stack of a thread consists of multiple segments on different clusters. In the case of a remote blocking call a new segment is created and the *locus of control* is moved to a new cluster. This view explains that value objects, local variables and parameters (all on the stack) always reside on the near cluster[10].

Note that each cluster has its own scheduler. Normal blocking and non-blocking calls execute on the processors of the cluster of their invocation. There is no automatic load balancing among clusters. Thus, threads only change clusters at a remote call explicitly indicated by the `@`-operator.

Remote non-blocking calls (indicated by both the `:-` and the `@`-operators) are simpler. A new child thread is set up on the appropriate cluster. Notice the difference between *subthreads* and *child threads*: a thread consists of one or more subthreads of which only one is active at the time, whereas child threads are threads on their own forked by some parent thread. Figure 7 shows the difference between blocking and non-blocking remote calls.

Note that although *exceptions* are not passed from child to parent threads, remote blocking calls behave exactly like ordinary routine calls with respect to exceptions. In a reliable system one might like to protect the program against exceptions caused by a remote thread (e.g. HW-failure on the remote cluster). Obviously one doesn't want to have the exception handler on the remote cluster as well, because everything on the remote cluster breaks in case of a HW-failure over there. The appropriate functionality may be accomplished by separating the remote non-blocking call into a local non-blocking call and a remote blocking call. More precisely, one writes a little stub routine that protects the remote call against every possible exception and terminates the local thread correctly producing a result indicating the error. Figure 8 shows a continously running program fragment with a watchdog thread on one cluster that starts a worker thread on a different cluster and monitors

[10] This freedom of dynamically moving the locus of control implies that a simple implementation should replicate all code on all clusters. However, one may imagine more sophisticated implementations with code caches on the clusters such that code is loaded on demand. This may be a better implementation on a machine with many clusters and little memory per cluster.

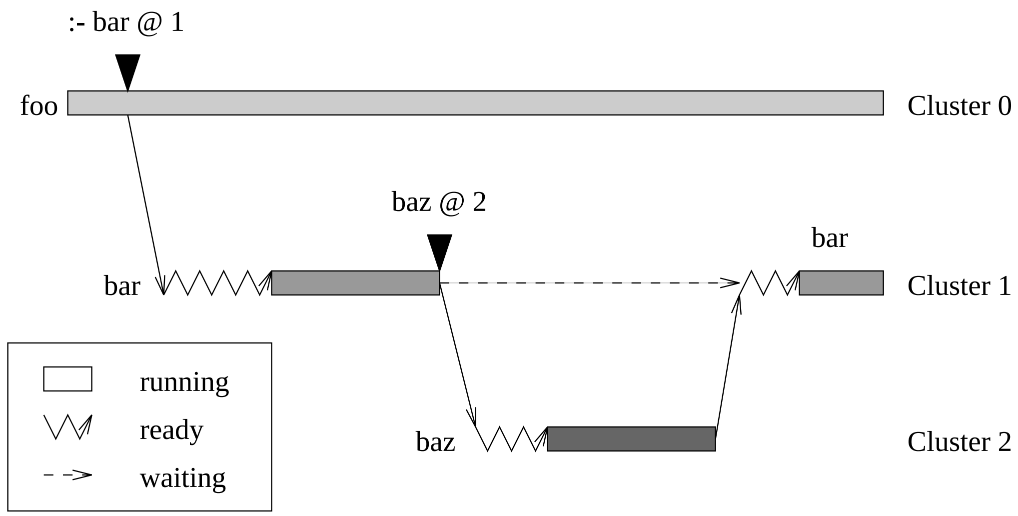

Figure 7: Blocking and non-blocking remote routine calls.

its execution. Programs like this are typical for control applications Exceptions in the worker thread are caught, passed to the watchdog where it causes the appropriate action (e.g. set an alarm and restart the worker on a different cluster).

4.4 Example: Fractals II

If we reconsider the fractals example from section 3.2 on a machine with multiple clusters, the algorithm gets slightly more complex because we cannot rely on the runtime scheduler to do the necessary load balancing between clusters. We have to explicitly assign each parallel computation of a pixel to a cluster. For a moderate number of clusters we can do this through a centralized controller[11] that keeps track of how many threads are active on each cluster. The class `CONTROLLER` (Figure 9) serves exactly this purpose. The attribute `free_clusters` is a gate that keeps track of free clusters in its internal list. `par_threads` counts the number of parallel threads to be forked before the controller should wait for a free cluster. The constant `threads_per_cluster` defines the maximum number of parallel threads per cluster. Its value should be slightly higher than the number of processors per cluster, such that all processors are kept busy and there is some spare work to hide the controller latency until a new thread is assigned to the cluster. `reset` initializes a parallel computation on this controller by resetting the `free_clusters` list and initializing `par_threads` to the maximum number of parallel threads on the whole machine. Tasks are submitted to the controller as bound routines. These tasks are wrapped into a simple routine that executes the task and returns the cluster number upon termination. These wrapper routines are started on a target cluster. There are two possibilities to determine the target cluster: If the initial number of parallel threads has not been reached, the thread is forked immediately equally distributed among the clusters. If the desired degree of parallelism is reached, `submit` awaits the first cluster to have a free slot and forks the thread there. `synch` just waits for all threads of this controller to terminate.

In the code of the class `FRACTALS` (Figure 10) we show one possible application of controllers. Note that we barely change the code in the `FRACTALS` class compared to the one cluster version in section 3.2. By employing the generic abstraction and encapsulation facilities of pSather we are

[11] Otherwise we need a distributed solution.

```
stub(worker: ROUT; cluster: INT): $EXCEPTION is
   protect
      worker @ cluster
   against $EXCEPTION then
      res:=exception;
   end;
end;

watch_dog is
   g::=GATE{$EXCEPTION};
   loop
      g:-stub(foo(1),1); -- start worker on 1
      g.take;            -- failure on 1
      g:-stub(foo(2),2); -- start worker on 2
      g.take;            -- failure on 2
   end;
endd
```

Figure 8: Exception handling across thread boundaries

able to successfully hide the issues of load balancing and architectural details (number of clusters, number of processors per cluster) from the programmer of the fractal application. This is a good example of how we envision parallel programming in pSather: `CONTROLLER` should be a library class and `FRACTALS` might belong to an application program.

4.5 Near and Far Objects

4.5.1 Reference Object Pointers

Each cluster manages a local heap of reference objects (`$OB`-descendants). A pointer to a reference object identifies both the cluster and the address within that cluster. We add the routine `o.where` to the standard reference object protocol in order to obtain the cluster id where the object is located. The access to attributes of an object `o` may either be *near* or *far* depending on whether the condition `o.where=CONFIG::current_cluster` is true or not, respectively. Note that each reference object is located as a whole on a single cluster with the exception of objects from the class `SPREAD{T}` (cf. section 6). The standard object protocol in pSather provides two predicates to check whether an object is near or far:

```
<obj-expr>.is_far
<obj-expr>.is_near
```

The predicate `is_far` returns true if the object is on a far cluster. `is_near` returns true if the variable references a near object (with respect to the executing thread). Void pointers return false to both `is_far` and `is_near`[12].

Implementation note: On machines with large virtual address spaces the memory management unit may generate traps when far pointers are dereferenced. In other implementations pSather

[12] In fact `void` is a perfectly valid pointer value, but it does not point to any object. Consequently, both predicates return false on `void` pointers. This convention is practical in many algorithms where we would like to operate on an object depending on whether it is far or near, and do nothing in the case it is void. It saves a lot of `void` checks.

```
class CONTROLLER is
   private attr free_clusters:#GATE{INT};
   private attr par_threads:INT;
   private const threads_per_cluster := 2; -- CM-5

   private wrap(r:ROUT):INT is
      -- wrapper routine, calls its argument and returns the
      -- the cluster on which it was executed
      r.call; res:=CONFIG::current_cluster;
   end;

   reset is
      free_clusters.clear;
      par_threads:=threads_per_cluster*CONFIG::num_clusters;
   end;

   submit(task:ROUT) is
      if par_threads>0 then
         par_threads:=par_threads-1;
         free_clusters:-wrap(task)
            @ par_threads.mod(CONFIG::num_clusters);
      else
         free_clusters:-wrap(task) @ free_clusters.take;
      end;
   end;

   synch is
      lock free_clusters.no_threads then end;
   end;
end;
```

Figure 9: CONTROLLER class

```
class FRACTALS is
     attr left, right, bottom, top, hstep, vstep: FLT;

     private man_pixel(p:ARRAY2{BOOL}, indices: {INT,INT}) is
        #(i:INT, j:INT):=indices;
        xpos::=i.flt*hstep+left; ypos::=j.flt*vstep+bottom;
        x::=xpos; y::=ypos;
        loop
           100.times!; -- Iterator
           x2::=x*x; y2::=y*y;
           if (x2+y2>=4.0) then p[i,j]:=true; return; end;
           x:=(x2-y2)+xpos; y:=x*y*2.0;
           p[i,j]:=false;
        end;
     end;

     mandel(p:ARRAY2{BOOL}) is
        -- Display part of the mandelbrot set in a 2-d black/white
        -- pixel array
        hstep::=(right-left)/asize1.flt;
        vstep::=(top-bottom)/asize2.flt;
        -- compute each pixel in parallel
        ctr::=#CONTROLLER; ctr.reset;
        loop ctr.submit(#ROUT(man_pixel(p, p.inds_tup2!))); end;
        -- ``inds_tup2'' is an iterator producing all index tuples
        -- for a 2-dimensional array
        ctr.synch;
     end;

  end;
```

Figure 10: `FRACTALS` class

pointer variables might consist of two words, one identifying the cluster and the other giving the address on that cluster. In these implementations the compiled code has to check on each pointer access whether a far memory request has to be issued.

4.5.2 `with-near` Statement

An obvious cost factor in the software implementation of the shared address space are the extra checks on pointers for near/far objects. Since the user may have knowledge about distribution of objects (e.g. certain components of an object are allocated locally), it would be useful if the user could specify that a pointer need not be checked for far reference.

We therefore add an assertion-like statement for the user to specify a set of variables to reference objects which are dynamically near to an executing thread. This **`with-near`**-statement has the following syntax:

near_stmt ⇒ **`with`** *ident_list* **`near`** *stmt_list [*`else` *stmt_list]* **`end`**

The list of identifiers may contain the following kinds of variables:

1. local variables
2. parameters
3. predefined variables: res, self

These *near variables* must be of a type derived from **`$OB`** (and henceforth, we will refer to them as near variables). The idea of near is not applicable to variables of either external or value types. The **`with-near`** statement works as follows. When the statement is encountered, the following assertion is tested: every variable in the identifier list references an object which is near to the current locus of control. If this does not hold the `else`-part is executed if available. An exception is raised if there is no `else`-part. The programmer is also asserting that in the first list of statements, the variables will be used to reference only near objects. The runtime system checks each assignment to one of the near variables to verify that the variables really reference only near objects at execution time. Note that one cannot define attributes of objects and shared attributes as near variables. This is because attributes and shareds may be accessed by more than one thread whereas locals and parameters always belong exclusively to one thread. Shared access makes run-time checks for the near assertion almost infeasible. There is a simple workaround for attributes to be defined as near variables by just assigning them to a local variable.

A typical situation in pSather is that you want to execute a routine such that accesses to the corresponding objects are near. There are two possibilities to do that. Either you move the locus of control to the object or you copy the object to the current cluster. Either solution may be appropriate depending on whether multiple copies of the object are acceptable or load balancing is less important than locality. The following examples show two code patterns with the proper use of the `with-near` statement:

```
bar:T is
   -- copy object to current cluster, then do bar
   with self near
      ...
   else
      res:=self.copy.bar;
   end;
end;

bar:T is
   -- move locus of control to object
   with self near
      ...
   else
      res:=self.bar @ self.where;
   end;
end;
```

4.5.3 Constructor Expressions for Remote Object Creation

Reference objects are created with *constructor expressions* in pSather. If not specified otherwise, all objects are created on the current cluster (`CONFIG::current_cluster`). In order to be able to create new objects on far clusters we need a way to locate object creation. We do this by adding the `@`-operator to the syntax of constructor expressions:

cons_expr ⇒ `#` *[type_spec] [*`(` *cons_elt (*`,` *cons_elt)*)* *] [* `@` *expr]*

As for remote calls the `@`-operator must be followed by an integer expression which evaluates to a valid cluster number. The following code fragment creates an array of 1000 integers on the lightest cluster according to some load balancing class:

```
block::=#ARRAY{INT}(asize:=1000) @ LOAD_BAL::lightest;
```

4.5.4 Shared Attributes

Like the code, space for *shared attributes* is allocated on all clusters. References to shared attributes always refer to the near instance of the shared attribute. Assignments to shared attributes only affect the near instance of a shared attribute. By combining assignments with the `@`-operator you can set any instance of a shared attribute selectively. Hence, the values stored in shared attributes may not necessarily be consistent throughout the machine. In order to broadcast a value to all instances of a shared attribute a special routine `bcast_x(T)` is implicitly defined for each shared attribute `x` of type `T`. `bcast_x(x:T)` broadcasts its argument (`x`) to all instances of a shared attribute and continues after the broadcast has been acknowledged by all clusters. Multiple active broadcasts to the same shared attribute are detected by the run-time system and raise an exception of type `BROADCAST_ERROR`.

4.6 Execution of Implicit Code

There are various cases when pSather programs have to execute implicit code. Since the locus of control is visible in the pSather programming model we must define where this code is executed:

Invariant clauses are executed at the location of the object they belong to. Thus the invariant `self.where=CONFIG::current_cluster` is always true.

Object attribute initialization expressions are executed on the cluster where the object is allocated.

```
class FOO is
   attr bar::=CONFIG::current_cluster
end;

...
   o::=#FOO;
   if o.where=o.bar then  -- is always true
...
```

`pre`-, `post`-, and `initial` expressions are executed in the context of the callee. This has subtle consequences for non-blocking and remote calls. Whether the precondition of `f` evaluates to true or not in the following example is subject to race conditions:

```
class FOO is
   attr bar:INT;
end;

...
   f(x:FOO) pre x.bar=1 is ... end;
...
   o::=#FOO(1);
   :- f(o);
   o.bar:=0;
...
```

We may use preconditions to test that a routine is always executed at the location of the corresponding object:

```
class FOO is
   f(FOO) pre self.where=CONFIG::current_cluster is ... end;
end;
```

5 Object Copying and Movement Operations

In a NUMA (Non-Uniform Memory Access) model such as pSather's, objects often need to be moved or copied to improve data locality. We demonstrate how the operations in Sather 1.0 can work together with the @-operator to provide copy/move mechanisms.

5.1 Regular, Deep and Near Copies

We first describe how the object-copy operations in Sather 1.0 (copy and deep_copy) work in pSather. Since these operations are routine calls, our use of the @-operator applies to them as well. We can have:

```
x.copy;
x.copy @ cluster_id;
```

In conformance with the semantics of remote routine calls (section 4.3), "x.copy" returns a local copy of x wherever the current thread executes (i.e. CONFIG::current_cluster). For "x.copy @ cluster_id", since the routine call executes at cluster_id, the result is a copy of x at cluster_id. The orthogonality of the object-creation operations and the use of @-operator therefore works out nicely. The @-operator allows the user to specify the final object's location independent of where the original object is located.

The routine deep_copy(ob:$OB):$OB (in SYS class) copies the graph of objects rooted at ob. Since the pSather semantics dictate that the location of the deep-copied object is where deep_copy is executed, the result object graph is located on one cluster even if the object graph was originally distributed over multiple clusters. Calling deep_copy with the appropriate cluster id using the @-operator allows us to specify where we want the deep-copied object to be.

Because of the distinction between remote and local objects, we feel that another copy routine (near_copy) is more useful in many cases. The routine near_copy(ob:$OB):$OB is defined in the SYS class. It copies the structure of all objects reachable from the root object ob directly via near pointers. The nearness is with respect to the root object and not with respect to the current locus of control. Suppose we have a root object O_1 which points to a remote object O_2 and O_2 points to another object O_3 which is near with respect to O_1. O_3 is not copied because it is not reachable directly via local pointers from O_1. The call SYS::near_copy(x) @ cluster_id copies all objects directly connected via near pointers to x, to the cluster given by cluster_id.

Implementation note: Since deep_copy and near_copy are part of a standard library, we expect that these operations will be implemented in an intelligent manner. When message startup cost is high, one possible strategy is to first pack the entire structure into a compact form, copying the compact form to the new cluster and then expanding it.

5.2 Migration of Objects

To support user-managed object migration, SYS class also defines a routine move_to:

```
move_to(ob:$OB; id:INT):$OB is
   res := ob.copy @ id;
   ob.destroy;
end;
```

The common usage will probably not require a client to use move_to with an @-operator (even though using an @-operator such as SYS::move_to(tree, current_processor) @ tree.where makes perfect sense). In a similar vein, we define the move-counterparts of near_copy and deep_copy:

```
near_move_to(ob:$OB; id:INT):$OB is
   res := near_copy(ob) @ id;
   -- And destroy all objects which have been copied.
end;

deep_move_to(ob:$OB; id:INT):$OB is
   res := deep_copy(ob) @ id;
   -- And destroy all objects which have been copied.
end;
```

The `x_move_to` operations return a new object identity. pSather in contrast to other approaches (e.g. Emerald [40]) does not support migrating objects which retain their identity, because the identity of an object is its address (including the cluster location) in our simple yet efficient model. The code for a class `$MOVEABLE` shows how the user can define a class for more transparently migratable objects.

```
class $MOVEABLE{T} is

  attr obj:T;
  move_to(cid:INT) is
      if (obj.where /= cid) then
         obj := SYS::move_to(obj, cid);
      end;
   end;
end;
```

It is not totally transparent because instead of:

```
x:POLYGON;
x.move_to(cid);
x.draw;
```

the user has to explicitly retrieve the object before invoking `draw`:

```
x:MOVEABLE{POLYGON};
x.move_to(cid);
x.obj.draw;
```

Thus far all the object allocation routines are either predefined or provided as part of a standard class (`SYS`). We expect these to handle most of the distributed cases, but we also need general mechanism (an example of which is the `PACKET` class described in the next section) that allows sophisticated users to define their own copying mechanisms.

5.3 General Copy/Move via PACKET

5.3.1 The Class PACKET

In general, one might want to have complex rules for deciding which parts of a data structure should be copied/moved to another cluster and which parts should remain uncopied/unmoved. An additional design constraint is that the copy/move of a large structure should execute efficiently. Due to high startup costs of communication, for the forseeable future this means that the most effective way of copying/moving data structures is to send packed representations. For this purpose, pSather provides a system class `PACKET` with the following interface:

```
class PACKET is
   -- Standard class for packing structures.
   attr psize:INT;
   mark(ob:$OB) is end;
   pack(ob:$OB, invalid_flag:BOOL):SAME is end;
   unpack:$OB is end;
   is_empty:BOOL is end;
end;
```

Instead of copying/moving the data structure, a user builds a `PACKET` object `p` which contains (in a packed form) the data structure to be copied/moved. The `PACKET` object (instead of the data structure) is copied/moved, and then it is unpacked at the destination to rebuild the desired data structure. The objects to be packed are marked using `p.mark(ob)`. Note that even though we use `mark`, the object remains unchanged. Rather it is the packet `p` that makes a note and remembers the object `ob`. The `p.pack(ob, flag)` function applied to an object `ob` will traverse all reachable objects from `ob`[13]. The object attributes are treated differently depending upon their type.

Value types: We distinguish between value types which inherit or do not inherit from the `BITS` class. Attributes whose type inherits from the BITS class are packed (i.e. copied into the packet). Otherwise, we have value objects with components. Each component is handled recursively depending on its type. (E.g. this handles the case when we have an attribute of value type with `$OB` sub-attributes.)

External type: Foreign attributes are not packed, and are replaced by the appropriate void value.

Reference or abstract type: If the attribute (near or not) contains a marked object, the object is recursively packed. Otherwise, the contents (pointer) is just copied into the packet[14].

The packed objects are destroyed when the `invalid_flag` parameter for `pack` operation is set to `true` . Therefore, the `invalid_flag` parameter gives the option of whether to retain or to destroy the original data-structure (and it has nothing to do with the packet).

`p.pack(ob, flag)` returns a `PACKET` object containing the packed objects that we want to copy or move. The objects are packed into a single block of memory, so that copying/moving a packet can make use of the machine's bulk communication mechanisms. Because the mark/pack/unpack operations work on each `PACKET` object, multiple threads, each with its own `PACKET` object, can perform packed copy/move in parallel. We can then invoke any of the copy/move operations on the returned packet using the `@`-operator (or not) appropriately. If we call `copy` on a packet, we can make use of the same packet to `copy` to several destinations. A moved packet is destroyed. These collapsed packets can be reconstituted by executing the `p.unpack` operation. A packet may be unpacked more than once. For example, if `p.pack` is called multiple times, then each call to `p.unpack` returns a copy of the corresponding graph of reference objects. `p.unpack` returns void when there is no more object to be extracted. `p.is_empty` returns true before any `pack` operation is called and after all objects are unpacked; otherwise it returns false. We do not expect the average user to do any of this, but the mechanism is needed for distributed data classes.

5.3.2 Implementation of PACKET

`PACKET` can just be an `ARRAY{CHAR}` or `ARRAY{INT}` with an additional hash-table for remembering objects. (After a `pack` operation, we might deallocate the hash-table, so whether we do a `copy`,

[13] After calling `p.pack(ob, flag)`, the packet does not remember any object. So even if we had called `p.mark(o)` and `o` was not one of the reachable objects in `p.pack(o1,flag)`, `p` no longer remembers `o`. The `ob` argument in `p.pack` may or may not be one of the marked objects.

[14] In some implementations (not the current CM-5 one), this may require transforming a near pointer to a far pointer.

near_copy, or deep_copy the effect is the same.) The unpack operation just looks at the array part to rebuild the data-structure and ignores the presence/absence of hash-table. The mark operation first creates a hash-table if none exists, and records ob into the table. If we copy a packet p without first calling p.pack, only the empty packet is copied without containing any objects.

5.3.3 Example of PACKET use

For example, a class for distributed binary trees might need a way of copying/moving a local subtree to another cluster, but leaving any pointers to other clusters unchanged. This would require a function that moved exactly those objects that are dynamically near. Every recursive call of tree_pack marks the subtree below, and eventually after the root has been marked, we pack everything that has been marked.

```
   private tree_pack(ch: PACKET) is
      if lt.is_near then lt.tree_pack(ch) end;
      if rt.is_near then rt.tree_pack(ch) end;
      ch.mark(self);
   end;
--  Footnote: We rely on the fact that void.is_near is false.

   tree_copy_to(id:INT):TREE is
      -- Suppose `self' is the root of sub-tree.  Return
      -- pointer to remote copy of sub-tree at cluster given
      -- by `id'.
      p:PACKET := PACKET::new;
      tree_pack(p);
      res := (p.pack(self, false).copy @ id).unpack;
   end;

   tree_move_to(id:INT):TREE is
      p:PACKET := PACKET::new;
      tree_pack(p);
      p := SYS::move_to(p.pack(self, true), id);
      res := p.unpack;
   end;
```

6 The Class SPREAD{T}, Replication and Reduction

6.1 The Class SPREAD{T}

Normal objects in pSather reside completely in the memory of the cluster they were created on. To support distributed objects we introduce the class `SPREAD{T}` as a base class for objects whose space is allocated by spreading over all clusters. As we proceed through the section we will show some essential classes based on `SPREAD{T}`.

We may allocate an object of class `SPREAD{T}` exactly like any other reference object in Sather:

```
s ::= #SPREAD{T};
```

If `T` is a reference type this statement allocates space for a pointer on each cluster of the machine. Otherwise, `T` is a value type and space is reserved according to the size of an object of type `T` on each cluster.

The use of `SPREAD{T}` is syntactically similar to the class `ARRAY{T}` in sequential Sather. The difference is that the index range is mapped on clusters instead of consecutive memory locations with the classic array. Thus, the statements with index expressions

```
local ::= s[CONFIG::current_cluster];
```

and

```
s[i] := local.copy @ i;
```

read the local instance of a `T`-typed object on the current cluster, and make a copy of a local object on cluster i and assign it to the element of `s` on cluster i, respectively.

In contrast to a regular Sather array where the index range corresponds to the size of the array defined at creation time, the indices of `SPREAD{T}` always range from 0 to `CONFIG` :: `num_clusters`-1.

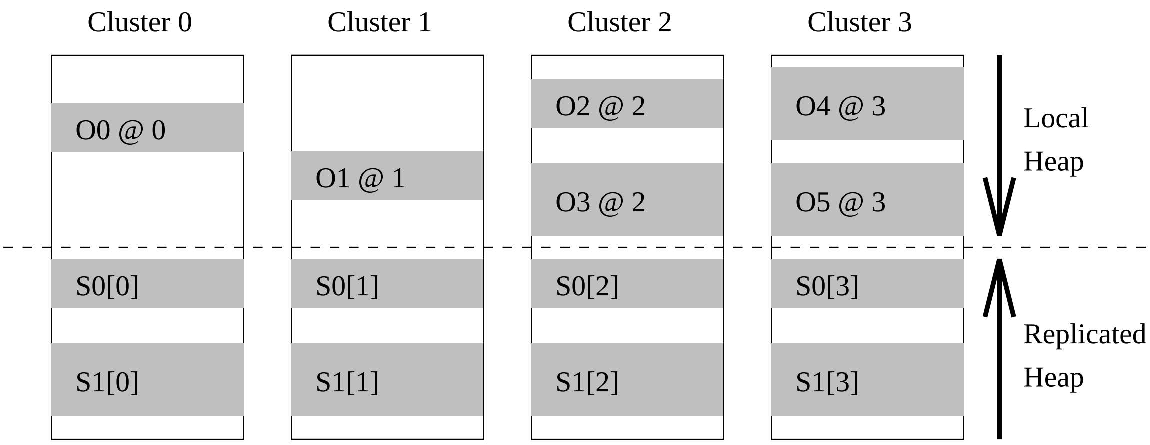

Figure 11: Memory organization for `SPREAD` and ordinary objects

Spread objects are a low-level concept in pSather and it is of central importance that remote parts of a spread object can be accessed directly without going through a centralized directory (mapping cluster indices to addresses of local chunks of memory) in order to avoid memory access bottlenecks at the directory. Therefore, spread objects need to share a single object identification on all clusters.

Implementation note: Since object identification means memory address in pSather, it may help to look at one possible implementation in order to understand the semantics of spread objects.

Ordinary reference objects are allocated from a heap that is managed for each cluster individually. In addition to this heap there is a second heap for spread objects, located in the same address range on each cluster. This heap is managed by a central agent for the whole machine. Thus, if new memory is requested for a replicated object a chunk of memory is reserved by the central agent in the replicated heap on the same address for each cluster. Figure 11 shows the memory map on each cluster after allocating six local (O0–O5) and two spread (S0, S1) objects. Note that the replicated heap may also hold other replicated entities like code, shared attributes and constants, whereas the stack segments for the threads are allocated in the local heaps.

6.2 Replication built on top of SPREAD{T}

Often we would like to replicate an object under the same name on all or a subset of clusters in order to have local access to the object irrespective of the locus of control. We may employ **SPREAD{T}** to implement a library class **REPL{T}** (Figure 12) providing the functionality to replicate an object of type **T**, access the local copy on any cluster, and to coordinate the copies when necessary.

This class allows reading (`local`) the local instance of the underlying spread object. In addition to that it provides a number of routines to broadcast a value over the whole set of clusters or just part of it. By passing the copy routine as a bound routine one is able to control the depth of copying when broadcasting. It is, for example, very useful to copy the objects and not just the pointers if `T` is a reference type. The copy routine may be anything as long as all arguments are bound and it returns exactly one result of the appropriate type, e.g.:

```
bcast(ROUT(root.copy))
```

The broadcast operations in Figure 12 do not guarantee that all the local instances in the replication range are consistent after a broadcast. Multiple simultaneous broadcasts may lead to inconsistencies because there is no synchronization. This is however sufficient in situations where consistency is not required or where control flow guarantees that only one broadcast is active at the time. If we want to guarantee consistency we need a class with synchronization such as **SYNCH_REPL** in the example below.

```
class SYNCH_REPL{T} is
    include REPL{T};

    attr mutex::=#GATE0;

    bcast(x:T, s:CLUSTER_SET) is
       lock mutex then
          loop [s.clusters!] := x; end;
          -- there are more elaborate methods
       end;
    end;

    bcast(x:T, r: RANGE, copy_rout: ROUT{T}:T) is
       lock mutex then
          loop c:=s.clusters!; [c] := copy_rout(x).call@c; end;
       end;
    end;
 end;
```

```
class REPL{T} is
   include SPREAD{T};

   bcast(x:T) is
      bcast(x, CONFIG::all_clusters);
   end;

   bcast(x:T, s:CLUSTER_SET) is
      loop [s.clusters!] := x; end;
      -- there are more elaborate methods
   end;

   -- copying broadcasts
   bcast(copy_rout: ROUT:T) is
      bcast(CONFIG::all_clusters, copy_rout);
   end;

   bcast(copy_rout: ROUT:T, s: CLUSTER_SET) is
      loop c::=s.clusters!; [c]:=copy_rout.call@c; end;
   end;

   -- tree copying broadcast to all clusters
   private tbc_helper(s,n:INT, copy_rout:ROUT{T}:T) is
      n2::=n/2;
      [s+n2]:=copy_rout.call([s])@s+n2;
      cobegin
         if n2>1 then :- tbc_helper(s,n2)@s end;
         if (n-n2)>1 then :- tbc_helper(s+n2,n-n2)@s+n2 end;
      end;
   end;

   tbcast(x:T, copy_rout: ROUT{T}:T) is
      [0]:=copy_rout.call(x)@0;
      if (CONFIG::num_clusters>1) then
         tbc_helper(0,CONFIG::num_clusters,copy_rout);
      end;
   end;

   local: T is
      res := [CONFIG::current_cluster];
   end;
end;
```

Figure 12: `REPL{T}` built on top of `SPREAD{T}`

6.3 Reduction built on top of SPREAD{T}

Another generally useful class can accumulate values in cluster-local accumulators according to a given reduction rule and finally reduce all the values on the clusters to one single value. This class is useful in a context where values are produced on many clusters and have to be reduced to a single value at the end.

```
class REDUCTOR{T} is

   include REPL{T};

   attr redex: ROUT{T,T}:T;
   attr cs: CLUSTER_SET := CONFIG::all_clusters; -- Default.
   attr z_val: T;

   reset is
      bcast(z_val, cs);
   end;

   accum(x:T) is
      [CONFIG::current_cluster] :=
         redex.call([CONFIG::current_cluster],x);
   end;

   result: T is
      res := z_val;
      loop i::=cs.clusters!; res:=redex.call(res,[i]); end;
      -- Here again: Better reduction strategies depending
      -- on the machine are welcome
   end;

end;
```

Note that the local accumulator in `accum` is not locked for mutual exclusion in this code. This works on machines with only one processor per cluster and non-preemptive semantics of threads on one processor. On architectures with more than one processor per cluster we need to add cluster local mutual exclusion across the `accum` routine. On the basis of this `REDUCTOR` class we can build all kinds of more specific reductors like `INT_SUM`, `FLT_SUM`, `INT_MAX`, `INT_MIN`, `FLT_PROD`, etc. We will better understand the usefulness of these classes in the context of distributed objects and data parallel computation in the next section.

7 The Built-in Class `$DIST{T}` and the `dist` Statement

On a parallel machine we would like to distribute data structures piecewise over the whole or part of the machine. In order to profit from local memory accesses we would also like computations to be located on the same processor as the involved data. In simple cases this is similar to the data-parallel model supported by many SIMD architectures, but pSather supports a much more general SPMD (single program multiple data) model of data parallelism.

7.1 `$DIST{T}` keeping track of different chunks of data

In general a *distributed data structure* consists of a *directory* and a number of *chunks*. The directory keeps references to all the chunks. The chunks hold the local parts of the distributed data structure. There may be more than one chunk per cluster but all data in one chunk are usually located on the same cluster. Figure 13 shows one possible memory organization for a distributed data structure consisting of a directory and five chunks on a machine with four clusters.

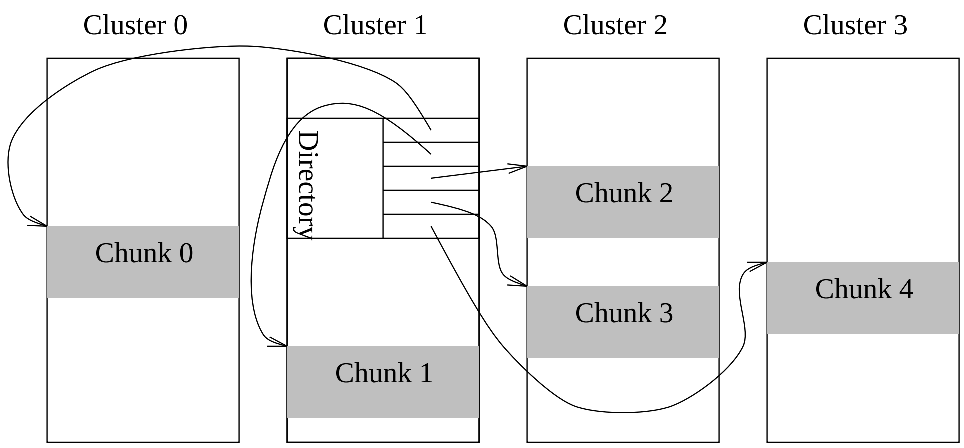

Figure 13: Memory organization for a `$DIST` object with its chunks

To support such distributed data-structures in a general way pSather incorporates a common abstract class `$DIST{T}`. `$DIST{T}` provides the basic information to support SPMD-like operations (cf. `dist`-statement), Section 7.2). The interface for the abstract class `$DIST{T}` is shown in Figure 14:

`nr_of_chunks`: Returns the number of chunks in this distributed data-structure.

`is_aligned_with`: **`aligned_with`** is a predicate checking whether two distributed objects are aligned with each other. To be aligned means that both objects have the same number of chunks and chunks with the same directory indices are located on the same cluster. Note that this implementation defines alignment between arbitrary `$DIST{T}` descendants. Nothing prevents you from aligning a distributed array of complex numbers to a distributed array of reals with the same size, for example. Depending on the details of the concrete distributed data-structure may allow for cheaper implementations of `is_aligned_with` (see implementation of `SIMPLE_DIST` (Figure 17, for example).

`chunks!`: The iter `chunks!` yields all chunks of a distributed data structure. It will be used by some implementations of the `dist`-statement.

```
abstract class $DIST{T} is
   -- supertype of all distributed data structures to be used
   -- with the dist-statement
   nr_of_chunks: INT;

   is_aligned_with(d:$DIST{$OB}): BOOL is
      -- check alignment
      if nr_of_chunks=d.nr_of_chunks then
         loop res:=chunks!.where=d.chunks!.where; until!(res=false) end;
      end;
   end;

   iter chunks!:T; -- iterate through all chunks in a given sequence

   iter chunks_on!(INT):T; -- iterate through all chunks on a given cluster
end;
```

Figure 14: Abstract `$DIST{T}` class

`chunks_on!`: The iter `chunks_on!` yields all chunks of a distributed data structure on a given cluster. It will be used by most implementations of the `dist`-statement.

Note that the abstract class `$DIST` merely implements the minimal features needed by the `dist`-statement. Concrete implementation of distributed objects provide a richer interface including operations to create, redistribute, add, and remove chunks from the distributed object. We will discuss a number of particularly useful implementations of distributed objects.

One possible simple implementation of `$DIST{T}` uses `ALIST{T}` to implement the directory (Figure 15). This is actually the data structure as drawn in Figure 13.

A more interesting implementation is built on top of `SPREAD{T}` with subdirectories on each cluster (Figure 16). This implementation is optimized for distributed structures with chunks on all or most of the clusters and big data-parallel operations. Note that a dist-statement can be executed distributed without having the directory as a bottleneck because each cluster holds its part of the directory. It could also serve as a base class for multiple parallel access distributed structures. The distributed directory could prevent a directory access bottleneck.

Even more simply we may provide a special class `SIMPLE_DIST` (Figure 17) with at most one chunk per cluster. This is a very common case for many distributed objects. Note that `is_aligned_with` gets particularly simple if two `SIMPLE_DIST{T}` objects are compared.

7.2 `$DIST{T}` and the `dist` Statement

pSather has a `dist`-statement for data-parallel computation on objects that are subtypes of `$DIST{T}`. In this section we will first introduce the syntax and the semantics of the `dist`-statement and show examples for the `dist`-statement in the next section. Syntactically the `dist`-statement is another block statement in pSather:

dist_stmt ⇒ `dist` *expr* `as` *ident (*, *expr* `as` *ident)** `do` *stmt_list* `end`

The `dist` statement executes its body in parallel *body threads*, one for each chunk of a given distributed object[15].

[15] Since the code in the body of the `dist`-statement is executed sequentially, whereas the single body threads are executed in parallel to each other, the body code determines the granularity of parallelism in a `dist`-statement.

```
class LDIST{T}<$DIST{T} is

   include $DIST{T};

   dir: ALIST{T}:=#ALIST{T};

   add_chunk(c:T) is
      dir:=dir.push(c);
   end;

   nr_of_chunks:INT is
      res:=dir.size;
   end;

   iter chunks!:T is
      loop res:=dir.elts!; yield; end;
   end;

   iter chunks_on(cl:INT)!:T is
      loop res:=dir.elts!; if res.where=cl then yield; end; end;
   end;

end;
```

Figure 15: Simple implementation of `$DIST{T}` based on a list directory

```
class SDIST{T}<$DIST{T} is

   include SPREAD{ALIST{T}};
   include $DIST{T};

   add_chunk(c:T) is
      if [c.where]=void then [c.where]:=#ALIST{T}@c.where end;
      [c.where]:=[c.where].push(c)@c.where;
   end;

   nr_of_chunks:INT is
      loop res:=res+[CONFIG::clusters!].size@c; end;
   end;

   iter chunks!:T is
      loop
         c::=CONFIG::clusters!;
         loop res:=chunks_on(c); yield; end;
      end;
   end;

   iter chunks_on(cl:INT)!:T
      if [cl]/=void then loop res:=elts! end; end;
   end;

end;
```

Figure 16: Concrete implementation of `$DIST{T}` with a distributed directory

```
class SIMPLE_DIST{T}<$DIST{T} is

   include SPREAD{T};
   include $DIST is_aligned_with($DIST{$OB}):BOOL -> o_is_aligned_with;

   nr_of_chunks:INT is
      loop if [CONFIG::clusters!]/=void then res:=res+1 end; end;
   end;

   is_aligned_with(d:$DIST{$OB}): BOOL is
      typecase d
         when SIMPLE_DIST{$OB} then res:=nr_of_chunks=d.nr_of_chunks
         else res:=o_is_aligned_with(d)
      end;
   end;

   iter chunks!:T is
      loop
         res::=[CONFIG::clusters!];
         if res/=void then yield; end;
      end;
   end;

   iter chunks_on(cl:INT)!:T is
      res:=[cl]; if res/=void then yield; end;
   end;

end;
```

Figure 17: Specialized implementation of `$DIST{T}` with a single chunk per cluster

The as expression in the header of the statement defines a *chunk variable* to relate distributed objects to variables referring to the corresponding chunks throughout the body of the dist-statement. The body thread is always executed on the cluster the corresponding chunk is located on. The purpose of this semantics is to bind parallel computation to the location of data, in order to exploit locality.

It is possible to specify more than one distributed object in the header, if all the distributed objects are pairwise aligned; i.e. x.is_aligned_with(y) is true for every pair of x and y in the header. An exception of type ALIGNMENT_ERROR is raised if the distributed objects are not all aligned[16]. All expressions before the as in the header must be of types descended from $DIST{T}. The chunk variables automatically get the type of the chunk, i.e. d as c, defines a variable c of type T where d's type is of a subtype of $DIST{T}.

```
plus(a:SAME):SAME
   -- A new vector equal to self plus a.
   pre is_aligned_with(a) is
   -- create result object
   res:=SAME::create;
   dist res as res_c, self as c, a as a_c do
      res_c.to_sum_of(c, a_c);
   end;
end;
```

Figure 18: Use of dist-statement in DVEC class

Before continuing with details of the dist-statement semantics, Figure 18 shows an example from the DVEC class for distributed vectors, illustrating the practical use of the dist-statement[17].

The DVEC class for distributed vectors is a $DIST{T}-descendant with sequential vectors as chunks. This is an important construction principle. Many distributed classes are built on top of their sequential counterparts by distributing sequential objects with the same functionality as chunks of a distributed data structure. The plus routine adds two distributed vectors and creates a new one as the result. The precondition for addition is that both self and the argument a are aligned to each other. Alignment for distributed vectors has the generic semantics for distributed objects, requiring that two objects have the same number of chunks and chunks are located pairwise on the same cluster (cf. section 7.1). Furthermore the vector chunks are required to have the same dimension pairwise.

After checking for alignment in the precondition, the routine plus continues by creating a new distributed vector of the same dimension as self for holding the sum. The actual computation is performed in the body of dist-statement over res, self, and a with chunk variables res_c, c, and a_c, respectively. Each body thread just uses the to_sum_of routine of ordinary vectors to sum over the single chunks. This is again a very typical pattern in distributed data structures: the distributed operation is just a distributed application of the ordinary operation.

The dist-statement is also a scope for *body local* variables. Local variables of a dist-statement body and the chunk variables defined in header are only visible within the body of the dist-statement. There is one instance of each body local variable per body thread.

The same instance of every variable in the surrounding scope is visible from the body threads of a dist-statement (i.e. local variables and parameters of the enclosing routine, self as an implicit

[16] Of course, it is always possible to refer to additional distributed objects within the body which are not aligned to the ones mentioned in the header.

[17] There are more examples from this class in section 7.3.

parameter and with it all attributes of the corresponding object, local variables of surrounding `dist`-statements, shared and constant attributes). Assignments to all these different classes of variables (except constant attributes) are allowed from within body threads, but note that the general rules for atomicity and consistency of memory operations (cf. section 3.5) apply in the context of the `dist`-statement under the assumption that the beginning and the end of a `dist`-statement correspond to implicit thread multi-fork and multi-join operations[18]. Thus, different body threads do not necessarily see a consistent picture of the shared variables unless they employ explicit synchronization operations like `lock`-statements around accesses to shared variables. Note, also, that the end of the `dist`-statement is a synchronization operation enforcing consistency in the sense of Section 3.5.2.

Implementation note: A naive implementation of the above semantics could lead to a completely sequential execution of a `dist`-statement because of the memory bottleneck at the location of the local variables of the surrounding scope. An important optimization is to pass variables that are only read in the body by value to each body thread. All accesses to these variables are then local. Note, however, that only pointers are passed for reference objects. For local access to the objects we need to replicate the objects first. For techniques to prevent read and write bottlenecks see sections 6 and 7.3.

The `dist`-statement belongs to the group of structured statements which have to perform a termination action. In the case of the `dist`-statement, proper structure termination means waiting for all body threads to terminate. This is very similar to the `cobegin`-statement in Section 3.1. This leads to the following semantic rules:

- If an exception is passed on beyond the end[19] of a `dist`-statement the statement waits for termination of all its body threads before passing the exception on to the next outer handler. If more than one body thread terminates exceptionally, only one of the exceptions is passed on beyond the end of the statement. One may look at the `dist`-statement as an implicit exception handler, that handles all exceptions by properly synchronizing all body threads and passing on one of the exceptions to the next outer handler.

- `return` from within a `dist`-statement terminates all body threads before eventually returning from the routine.

- `yield` statements in the body of a `dist`-statement and iter-calls from the body of a `dist`-statement are not allowed.

In addition to structure termination issues we may have multiple parallel body threads changing the same local variable. This implies some restrictions on the use of these variables in statements relying on established assertions.

- `with-near`-statements in `dist`-statements may not refer to local variables which are defined outside the `dist`-statement but are assigned in the body.

Note that it is in theory possible to modify the distribution structure of a distributed object from within the body of a `dist`-statement. Doing so may lead to unpredictable behavior of this `dist`-statement.

[18]Note that this is not exactly the same termination semantics as in the `cobegin`-statement (Section 3.1), where the termination of all threads forked within the body have to terminate before the whole statement terminates. In the `dist-statement` only the body threads need to terminate for termination of the whole statement. Threads forked by the body threads may continue running beyond termination of the `dist`-statement.

[19]We say that an exception is passed on beyond the end of a statement if it was raised (explicitly by the `raise`-statement, or implicitly by an inner exception handler that could not catch the exception or the runtime system) within the body of that statement and cannot be handled in the body.

7.3 Examples with the `dist` Statement

First, some examples from the **DVEC** class for distributed vectors. **DVEC** is based on **DARRAY{FLT}** and provides the same functionality as the VEC class in serial Sather library.

In the first example we multiply each element of the vector by a scalar factor. This example already shows that it is important to pass read-only variables (**s** in the example) by value to the parallel body threads. If we didn't do that in this case the memory access bottleneck to **s** would sequentialize the whole computation.

```
scale_by(s:FLT) is
   -- Scale self by s.
   dist self as c do
      loop c.set_elts!(a*c.elts!) end
   end;
end;
```

The memory access issues get even more complex for the dot product of two vectors. Recall the general reductor class we introduced in the last section built on top of the **SPREAD** class. This is exactly the intended use for this class: First we create a reductor object with a reduction operation (**FLT::plus** in this case), a reduction cluster set equal to **self.cl_set**, the cluster set occupied by chunks of the distributed vector, and a zero value 0.0. Note that all accesses to the reductor **red** through **accum** are local reads and writes because the reductor is a replicated object. The only place where communication is necessary, is for forking the body threads in the **dist**-statement and for collecting the results in the **result** routine of the reductor.

```
dot(v:SAME):FLT
   -- The dot product of self and v.
   pre aligned_with(v) is
   -- create reductor on the same cluster set as self
   red::=#REDUCTOR{FLT}(#ROUT(FLT::plus(_,_)), cl_set, 0.0);
   dist self as c, v as v_c do
      loop red.accum(c.elts!*v_c.elts!) end;
   end;
   res := red.result;
end;
```

In the next example we look for the index of the greatest element in the array. The algorithm first looks for the maximum and its index in each chunk. Each chunk knows its index offset in the vector (**offset**). This is an example where we use a gate to guarantee mutually exclusive access for the parallel body threads when they update the maximum. Note that **res** and **mval** are shared by all body threads. This together with the locking of the **mutex** gate may lead to write contention in the final stage of each body thread. An alternative solution would have been to use a reductor object to do the maximum reduction. The trade-off is whether the parallel body threads terminate sufficiently out of synch. With the reductor we wait until all body threads are terminated and do the reduction afterwards (Asynchronous reduction with more synchronization overhead vs. synchronous solution).

```
max_index:INT is
   -- The index of the maximum component of self. Lowest index
   -- in case of equality.
   mval::=[0]; -- initialize global maximum with first vector element
   res := 0; mutex::=#GATE0;
   dist self as c do
      lmax::=[c.offset]; li::=c.offset;
      loop
         i::=inds!; le::=elts!;
         if le>lmax then lmax:=e; li:=i end;
      end;
      lock mutex then
         if lmax > mval then mval:=lmax; res:=li end;
      end;
   end;
end;
```

The next example uses even more complex synchronization between the parallel body threads. In `bounded_square_dist_to` it would be very inefficient to compute the complete square of the distance if the bound is reached very quickly. Therefore we update the total square distance after every 100th iteration of the loop and stop the computation as soon as the bound is reached. In the worst case we do 100 additional iterations in the body thread before terminating if the bound is reached. The maximum value for the counter needs to be balanced against the overhead for extra computation vs. communication on the architecture for which the library is designed.

```
bounded_square_dist_to(v: SAME, sbnd:FLT):FLT
   -- The square of the Euclidean distance from self to v if it is
   -- less than or equal to 'sbnd', '-1.0' if it is greater than it.
   -- Can avoid some operations if used in a bounding test.
   pre aligned_with(a) is
   mutex::=#GATE0;
   dist self as c, a as a_c do
      lsum ::= 0.0; ct ::= 0;
      loop
         lsum := lsum+(c.elts!-a_c.elts!).square; ct:=ct+1;
         if ct >= 100 then
            lock mutex then res := res+lsum; end; -- atomic update
            ct:=0; lsum:=0.0;
            if res > sbnd then break end;
         end;
      end;
      lock mutex then res := res+lsum; end; -- atomic update
   end;
   if res > sbnd then res:=-1.0 end;
end;
```

One typical usage of the class `REPL{T}` is to provide an object for local access in all parallel body threads of a `dist`-statement. Consider the following excerpt of a graphics program that applies a transformation matrix to all vectors of a large structure, in order to rotate and translate a three-dimensional picture:

```
-- PICTURE: some distributed structured collection of points
pic: PICTURE;
a: T_MATRIX;

-- replicate transformation matrix
t::=#REPL{T_MATRIX};
t.bcast(#ROUT(a.copy));
dist pic as pic_c do
   loop p::=pic_c.points; p.transform(t.local); end;
end;
```

This example would produce exactly the same result without replicating the transformation matrix. However, since all body threads would refer to the same physical location of the transformation matrix, the bottleneck would slow down the program to completely sequential execution.

7.4 The sync-Statement

A common pattern of synchronization among body threads is barrier synchronization. pSather provides a special **sync**-statement for barrier synchronization of body threads with the following syntax:

sync_stmt ⇒ **sync**

sync-statements must be syntactically enclosed by a **dist**-statement. When one of the body threads encounters a **sync**-statement it waits until all other **body**-threads of the same **dist**-statement have arrived at a **sync**-statement. As soon as all body threads are waiting on a **sync**-statement the barrier is released and all body threads continue after the **sync**-statement. If one of the body threads encounters a **sync**-statement while a partner body thread has already terminated or if a body thread terminates while others are waiting at a barrier, a runtime error occurs.

One typical application of the **sync**-statement are data-parallel computations that proceed in multiple phases, wherby each phase has to be finished before the new one is started. Parallel merge sort algorithms are one typical example. Figure 19 implements a coarse-grain variant of a parallel sorting network. It sorts the whole vector by first sorting each chunk an then performing a parallel merge of all chunks. The parallel merge proceeds in $\frac{\lceil \lg^2 n \rceil + \lceil \lg n \rceil}{2}$ parallel merge steps merging each chunk with another chunk in the vector, whereby n is the number of chunks. For the details of how to determine the partner chunk in the merge steps we refer to the sorting network introduced in [17, Chapter 28]. The merge steps have to proceed in lock-step. Each new merge step depends on the results of the preceding merge steps. The merge steps proceed in two phases: First, the local chunk is merged with a remote partner chunk into a temporary vector chunk (**t**). After all body threads have completed this operation the body threads encounter a barrier (first **sync**). In the second phase, the local chunk is exchanged with the temporary chunk (double buffering) in all body threads and the body threads synchronize again (second **sync**).

7.5 The Class $MDIST{T}

In all the previous sections there was only one **dist**-statement with its parallel body threads operating on the distributed data structures. The only form of parallelism was non-nested **dist**-statements. This corresponds to the simple SPMD (single-program multiple-data) model. For many applications this is all we need. There are, however, applications with multiple parallel body threads accessing the same distributed data-structure (e.g. data base). This creates a number of additional problems:

- To preserve consistency of the data the chunks must observe a single writer multiple reader protocol (cf. section 3.4 for a readers/writer library class).

```
merge_with(i:INT, into: VEC_CHUNK) is
   -- merge local chunk with chunk on cluster i and keep the smaller
   -- (larger) values depending on the chunk positions
   if i<CONFIG::num_clusters then -- only merge if there is a partner
      l:VEC_CHUNK:=self[CONFIG::current_cluster]; r:VEC_CHUNK:=self[i];
      if i>CONFIG::current_cluster then -- merge from bottom
         lj:INT:=0; rj:INT:=0;
         loop j:INT:=0.upto!(into.asize-1);
            if l[lj]<r[rj] then into[j]:=l[lj]; lj:=lj+1;
            else into[j]:=r[rj]; rj:=rj+1; end end;
      else -- merge from top
         lj:=l.asize-1; rj:INT:=r.asize-1;
         loop j:=(into.asize-1).downto(0);
            if l[lj]>r[rj] then into[j]:=l[lj]; lj:=lj-1;
            else into[j]:=r[rj]; rj:=rj-1; end end end end end;

xchg_with(c: VEC_CHUNK) is
   -- exchange c with the local vector chunk
   t:VEC_CHUNK:=[CONFIG::current_cluster];
   [CONFIG::current_cluster]:=c; c:=t end;

sort is
   dist self as c do
      c.sort; -- inherited from VECTOR
      m:INT:=1; -- merge the sorted chunks in parallel
      t:VEC_CHUNK:=VEC_CHUNK(offset:=c.offset,asize:=c.asize);
      loop while!(m<CONFIG::num_clusters); -- while! is also an iterator
         m:=m*2;
         k:INT:=(i/m)*m; -- k := lowest multiple of m lesser than i
         merge_with(k+(n-i-1).mod(m),t); sync; -- merge & sync
         xchg_with(t); sync; -- set new chunks & sync
         n:=m/2;
         loop while!(n>1);
            k:=(i/n)*n; -- k := lowest multiple of n lesser than i
            merge_with(k+(i+n/2).mod(n)); sync; -- merge
            xchg_with(t); sync; -- set new chunks & sync
            n:=n/2; end end end end;
```

Figure 19: Use of `dist`-statement for the `sort` routine in `DVEC`

- Each client of a distributed data structure should have his own copy of the directory. Otherwise, the directory accesses will lead to a severe memory bottleneck on the cluster holding the directory in a large system.
- The copies of the directory need to be kept consistent if a client wants to change the directory.

Changes of the distribution structure should happen only rarely in well designed distributed data structures. Normally only the size of the chunks grows and shrinks corresponding to the size of the structure. The number and location of the chunks should stay the same throughout the lifetime of a distributed data structure.

The following library class `$MDIST{T}` provides a generic solution for distributed structures being accessed from multiple threads. This is just one example of the kind of classes that can be as a descendant of the built-in class `$DIST{T}`. The routine `dup_header` copies the header information to a new client thread. The object keeps track of its header duplicates in a cyclically linked chain of headers. The header itself is a `RW_SYNC`. Users should therefore bracket each operation reading the header (virtually everything) by a `start_read/end_read`-pair. `start_hchange` and `end_hchange` serve to enclose changes to the header. `start_hchange` locks a one-per-distributed object gate `wlock` and then starts a write access to all headers in the chain. Note that `wlock` is not duplicated by `set_header`, keeping a single copy for all headers of the same object (collection of chunks). The driving assumption behind this design is that header changes happen very infrequently compared to read accesses to the header (directory, etc.). Therefore opening a header for a read operation is very cheap (access to one or two local gates, see `RW_SYNC`), whereas locking all headers for write might be pretty expensive with many headers.

```
class $MDIST{T}<$DIST{T} is
   include LDIST{T};
   include RW_SYNC;
   attr wlock::=#GATE0;
   attr chain::=self;

   private set_header(from: SAME) is
      -- maintain a ring of copies
      dir:=from.dir.copy; readers:=#GATE0; writer:=#GATE0;
   end;

   dup_header:SAME is
      res:=copy; res.set_header(self); res.chain:=chain; chain:=res;
   end;

   start_hchange is
      lock wlock then
         c::=self; loop c.start_write; c:=c.chain; while(c/=self) end end;
   end;

   end_hchange is
      h::=chain; -- next
      cobegin -- maybe it's worth to do it in parallel
         loop while(h/=self); :-h.set_header(self)@h.where; h:=h.chain; end;
      end;
      h:=self; loop h.end_write; h:=h.chain; while(h/=self); end;
   end;
end;
```

7.6 Other Distributed Data Structures

`$DIST{T}` is not restricted to such simple data structures as arrays. One could for example build distributed balanced trees where each chunk holds a subtree of the whole tree. The top-most nodes of the tree are replicated in each chunk. The idea is that most operations affect only nodes close to the leaves in a large tree. On the other hand the top-most nodes of a tree are the most read parts of a tree because any operation starts at the root. The balancing algorithm keeps all subtrees (chunks) about the same size which also leads to good load balancing. Preliminary versions of tree abstractions have been studied, but more work remains to be done.

Furthermore we can implement multi-dimensional arrays, lists, tables, hash tables, vectors, matrices, etc. on top of the `DARRAY` abstraction. Actually we took many of the examples in section 7.3 from a class for distributed vectors.

We think that the `$DIST` class together with the `dist` statement are very powerful tools to create parallel library classes optimally utilizing the power of the underlying machine without bothering the user with the architectural details. The next question to be answered will be whether there is a set of data structure classes with the following properties:

1. They are as a whole general enough to cover most users' needs for parallel computing.
2. They are special enough that there exist efficient solutions.
3. They are independent enough from specific architectures to be portable to many different machines.

`$DIST{T}` should be the common ancestor for this class library.

So far we have used the `$DIST{T}`-abstraction for distributed arrays and matrices, distributed hash-tables and distributed schedulers for discrete event simulation.

8 Related Work

This section gives the design dimensions for parallel object-oriented languages and describes where pSather fits in. It also goes into some detail in comparing parallel constructs in pSather with other approaches. This includes:

- high-level synchronization constructs (gates vs. monitor [34], M-structures [8])
- constructs to support NUMA (pSather's @-operator and copy/move operations vs. object movement in Emerald [40])
- constructs to support data-parallelism (`dist`-statement and `$DIST{T}` class vs. approaches in PC++ [30], [46] and C* [35])

A more complete and detailed description of related work (which involves discussing other specific parallel object-oriented languages) can be found in [48].

8.1 Processes/Threads

There are three general ways to create parallel processes or threads in object-oriented languages.

- A thread may be explicitly treated as a first-class object in the system (e.g. Presto [11]). In this case, methods are defined in the threads class to activate, suspend and perform other synchronization operations (e.g. fork-join). In this model, threads are independent of data objects and multiple threads can execute on an object in parallel.
- A second approach is to have active objects (*actors* [2]), each with its own message queue and thread of control. From the user's point of view, a thread exists to receive and service incoming message requests. Examples include POOL2 [4] and ABCL [24].
- In a third approach, threads of control are independent of any object in the system and managed by the system (e.g. scheduling). In this model (e.g. Hybrid [59], COOL [15]), objects are passive while threads are the loci of control. Normally, the programmer does not have any explicit handle to the threads, so that he/she cannot perform operations like moving the thread object from one scheduler to another. Most languages in this model allow multiple parallel threads to execute in an object. Some (e.g. Hybrid [59]) group objects into protection domains such that at most one thread can be active in a domain.

The design choice affects how threads and objects are created. For example, if the thread is an explicit object, to create and schedule a thread, one will invoke methods in a `THREAD` class. This approach also requires the language support routines or some form of closures as first-class objects, so that a created thread can use the value of the routine/closure to decide what to start executing.

On the other hand, in the active object model, a new thread is implicitly created and becomes active whenever an object is created. So instead of introducing thread creation construct, such a language needs a way to decide which messages can be received. In POOL2, each class definition has a *body* code (e.g. POOL2). An active object, when created, starts executing the body code that decides which incoming messages can be received.

In the system-managed thread/passive object model, thread creation is usually decoupled from object operations because code blocks and/or routines are not first-class objects. Languages with this model have constructs that support explicit thread creation (e.g. the reflex operation in Hybrid [59] or invocation of *parallel* function in COOL [15]).

pSather currently follows the third model. The threads are not however completely invisible to the user. For example, a thread can be attached to a gate ("`g :- f`") and by testing the predicate `g.no_threads`, the user can determine if the thread executing `f` has terminated.

```
abstract class $THREAD{T} is
  fork(ROUT) is end;
  -- The call ``fork(#ROUT(o.f))'' would be equivalent to
  -- ``:- o.f'' in current design.

  fork_at(ROUT, cluster_id:INT) is end;
  -- The call ``fork(#ROUT(o.f),i)'' would be equivalent to
  -- ``:- o.f@i'' in current design.

  fork(g:$GATE{T}, r:ROUT) is end;
  -- The call ``fork(g,#ROUT(o.f))'' would be equivalent to
  -- ``g :- o.f'' in current design.

  fork_at(g:$GATE{T}, r:ROUT, cluster_id:INT) is end;
  -- The call ``fork(g,#ROUT(o.f),i)'' would be equivalent to
  -- ``g :- o.f@i'' in current design.
end;
```

Figure 20: **THREAD{T}** class

One reason that pSather does not treat threads as first-class objects is that an older version of Sather [60] does not support any form of closure or routine as first-class objects. The new language specification has a form of closure called bound routines (section 2.4). As pSather evolves, one might imagine consolidating the deferred assignment with the normal class semantics by supporting a predefined **THREAD{T}** class (Figure 20). On the other hand, the use of a distinct construct like "`:-`" helps to clarify programs and allows a user reading a program to easily pick out the code that departs from normal sequential execution.

A design goal in pSather is suitability for efficient implementation. We therefore do not adopt the actor model [2] because of the performance costs of maintaining a message queue for each object, and disallowing parallel operations (e.g. reads) on an object.

In this design, the object-oriented term "message passing" in pSather does not involve communication between threads but has procedure invocation semantics instead; we might view it as message-passing between passive objects rather than threads. pSather does not need message-passing forms of parallel constructs like actors in POOL2 [4], broadcast in Orca [6], or asynchronous reply in Natasha [20] and ConcurrentSmalltalk [67]. In pSather, sequential routine calls are viewed as the default synchronous mode of message-passing while the deferred-assignment statement corresponds to asynchronous message-passing.

8.2 Machine and Programming Model

Some languages are more suited to certain architectures than others. For example, languages such as Orca and Distributed Smalltalk [9] are aimed at distributed systems. There is no shared address space like pSather, so that an object is not directly accessible from all processors. To share objects among processes, some languages (e.g. Orca) provide a *shared data-object model*, in which a parent process can pass its objects to its child processes via `shared` parameters in the children. Thus the shared objects serve as communication channels in a machine model where each processor has a logically distinct address space.

Other languages such as μC++ [14] and parallel versions of Eiffel implicitly assume a uniform shared address space. The clustered machine model in pSather (section 4.1) is one of the more distinct departures from other parallel languages. It provides a flexible model for both shared-memory multiprocessors and distributed-memory architectures whose network latencies are of the order of a few hundred (or fewer) instructions. We feel that this is a justifiable choice because many parallel architectures are converging to this characterization, and scalability and programmability considerations have led to many efforts at supporting distributed shared memory with NUMA characteristics.

8.3 Synchronization

There are two general approaches to achieving synchronization among threads [5] — shared data or message-passing. An example of the shared data approach is the use of monitors ([38], [66]) in Concurrent Pascal ([34]) and Mesa [42]; on the other hand, the message-passing approach is used in notations such as the rendezvous in Ada ([7], [65]), channels in CSP (Communicating Sequential Processes [39]) and Occam [62], and the send/receive constructs in PLITS [26].

In an object-oriented language, conceptually, objects interact among themselves by message passing. An object invokes a routine (or method) on another object or itself by sending a message to the destination object. It would therefore seem natural that object-oriented languages should adopt a message-passing approach for synchronization. This is indeed the approach adopted by the actor languages such as POOL2. Synchronization is achieved by controlling the receipt of messages.

Message-passing however is not the only means to achieve synchronization in parallel object-oriented languages. In the model with explicit thread objects (e.g. PRESTO), synchronization is more in the tradition of well-understood constructs such as fork-join and locks except that these mechanisms are not built-in and are defined as part of a class interface.

The synchronization mechanisms in the system-managed thread/passive object model depend on whether multiple parallel threads can execute in an object. If this is the case, explicit synchronization mechanisms are needed. The difference from the PRESTO-like model is that the synchronization constructs are normally provided via additional language extensions. Consider each of the synchronization patterns (lock protection, barrier and conditional wait) in COOL:

- To achieve locking, attributes and functions can be qualified as mutex at declaration.
- There is a predefined `binc` construct which encloses a block of statements. The current thread suspends until all threads created during the execution of the block terminate.
- There is a predefined class `cond`; objects of this class provide the functionalities of a condition variable.

In pSather, all three synchronization patterns can be done using gates (though the `cobegin-end`-statement provides a cleaner way to express barrier synchronization).

When multiple threads cannot execute on an object, locking is automatically available. But it is not clear (e.g. in the case of Hybrid) how the condition-wait and barrier synchronization patterns can be easily achieved without additional constructs.

8.3.1 Comparison of Synchronization Constructs

In pSather, since multiple threads can execute on an object, additional constructs are defined for synchronization. Instead of providing a different construct for each commonly-used synchronization (e.g. lock vs. barrier vs. conditional wait), a gate object unifies all the common synchronizations. The philosophy behind the design of `$GATE` classes resembles other synchronization constructs in other languages, such as monitors in Mesa and Concurrent Pascal [34], and M-structures [8] in Id, such that a relatively small set of powerful operations are provided, on top of which the user can build more sophisticated synchronization mechanisms.

Monitors. The pSather gate objects were previously called monitors [27] because of a similarity with the monitor concept in Mesa [42] and Concurrent Pascal [34].

Firstly a gate operation (monitor entry procedure) guarantees a thread (process) exclusive access to the object (module). But in pSather, the gate operations are predefined since `GATE{T}` and `GATE0` are predefined classes. In Mesa (for example), the entry procedures are user-defined because any module can be declared to be a monitor. This means that if a Mesa monitor operation is ever suspended, the user has to take care that it gets resumed correctly later and to prevent deadlocks.

In Mesa, condition variables can be declared in monitors such that a `wait` operation on a condition variable atomically suspends the process on a queue while a `notify` resumes one of the processes suspended on the condition variable. This functionality is achieved in pSather by the `take` and `enqueue` operations (corresponding to `wait` and `notify` respectively).

However, in terms of programming style, gates and monitors are used quite differently. We expect gates to be mostly used as components of objects to control synchronization among different routine calls acting on the same object. On the other hand, Mesa monitors are roughly "protected classes": "classes" because a monitor is an instance of a module (which encapsulates both data and procedures in a unit); "protected" because a monitor entry procedure is guaranteed exclusive access to the monitor. The `wait` and `notify` operations are also lower level than gate operations.

The deferred assignment unifies the functionalities of a gate with thread creation. As a result, a gate can be used as a future (section 3.3.4). This is not possible in Mesa because the monitor functionalities are independent of process creation.

M-structures. M-structures are designed to overcome the absence of state in Id, by allowing data structures to be updated. An M-structure has a state (*empty* or *full*) associated with it. This is like the bound state of gate objects and is used to synchronize access to M-structure. There are only two primitive atomic `take` and `put` operations that can be performed. When doing `take` on an empty M-structure, the thread is suspended, until another thread performs a `put` and atomically stores the value into the M-structure. The stored value is atomically retrieved by `take`. This is similar to the `take` and `set` operations in gates.

Unlike gates, M-structures do not store multiple values; thus there is no need to have any `enqueue` operation. The `examine` operation (similar to `GATE`'s `read`) is built on top of `take` and `put`.

```
def examine c = { v = take c ;  -- 'c' is an M-structure.
                  _ = put c v;
                In   v }
```

8.4 Object Placement

One attractive feature of object-oriented programming is that objects provide a high-level abstraction for programmers to deal with memory (shared or distributed). Because (local/remote) memory latency costs are more critical than costs of computation cycles, the user view of objects is especially relevant for program efficiency. We will treat the two aspects of placement of objects — allocation and relocation — together.

Some languages have a uniform shared memory model, so that if implemented on distributed memory multiprocessors, the communication costs of remote access are hidden from the programmer, and a runtime system (e.g. Tarmac [52]), that automatically performs load balancing and maps allocated objects on different processors, is needed.

There are however parallel object-oriented languages which support a non-uniform shared address space in a high-level manner. For example, Sloop [51] allows the programmer to specify *alignment* relationships among objects via calls to an *align* routine. The alignment relationship specifies the "spatial" relationship of objects; for example, two strongly aligned objects are always located on the same processor so that whenever one is moved to a different processor, the other has to move with it. Calls to the align routine may cause objects to be moved.

Similarly, Emerald [40] and pSather incorporate object placement in the language semantics. In fact, object mobility is a major design goal and affects the language design (e.g. parameter passing) of Emerald.

Emerald's location-independence vs. pSather's `@`-operator. Emerald supports location-independent invocation; a routine invoked on an object always executes at the object's processor, wherever the invoking thread might be. When a routine is invoked on an object, it is executed at the object's processor. Therefore when an object moves, activation records of its routines have to be relocated as well. This means that the runtime system has to keep track of the activation records of every movable object; this might entail high runtime costs. Jul et al [40] describes how to reduce such costs and what to update when activation records are moved. In pSather, the locus of control is independent of the object's location and is specified by the `@`-operator. The `@`-operator has the relatively simple semantics of specifying where a subthread executes. The orthogonality of the design allows this semantics of `@`-operator to be used with the copy/move operations for relocating objects. By using the `@`-operator, a thread's control can span multiple clusters even though subthreads (or activation records with locations) stay on a fixed cluster.

Emerald's transparent move vs. pSather's move/copy operations. The movement of Emerald objects is transparent to the user. Emerald distinguishes between objects which can be moved (*global objects*) and those which cannot be moved (*local objects*), and implements them differently. A global object is not referenced directly by any user variable. User variables point directly only to local objects or local *object descriptors*. An object descriptor either holds a pointer to a resident global object or a forwarding address which gives the processor on which the global object is resident. When an object is moved, its object descriptors on the source and destination processors are updated accordingly.

PSather's move operations do not relocate objects in a transparent manner. The programmer has to update the references to the new object. But pSather does allow a programmer to implement objects which move in an almost transparent manner. (The `$MOVEABLE{T}` class in section 5 corresponds to the object descriptor in Emerald.) There is also a variety of copy/move operations that allow a user to have complex rules to decide which parts of a data structure should be copied/moved.

8.5 Support for Data-Parallelism

In pSather, there is a `dist`-statement which specifies that its body is executed for each chunk in a distributed data structure (whose type is a descendent of `$DIST{T}`). There are five characteristics to note for pSather's style of data-parallelism.

- The so-called "data", which are operated upon in parallel, are actually large-grained chunks (e.g. tree, array) consisting of finer-grained data (e.g. tree nodes, array elements).
- Although data-parallelism was previously associated with the execution mode of SIMD machines [3], the `dist`-statement moves away from this association to an SPMD form of data-parallelism in which parallel threads execute the same piece of code on different chunks, but not necessarily in lock-step.
- The execution of body code is co-located with the cluster location of the corresponding chunk to ensure data locality. From the user's point of view, load balancing then consists of partitioning the distributed data structure such that each chunk requires approximately the same amount of computation.
- The setup can also be used for dynamic distributed data structures which can grow or shrink during program execution.

- A distributed object is just like any ordinary object and is not used exclusively only in `dist`-statements. For example, it is possible to perform the usual object operations (e.g. routine calls, iterator calls) on distributed objects just like ordinary objects.

In C* [35], the user can declare a *domain* data type (just like a C struct) and then use this data type to declare domain arrays. A domain array is then used in a *domain select* statement such that the body of the domain select statement is executed in parallel on every element of the domain array. Comparing C* with pSather:

- The data operated upon in parallel in C* is fine-grained, compared to the large-grained chunks in pSather.
- The execution of the body is synchronous so that the model of computation is still SIMD.
- Like pSather, the execution of the body code is located on the same processor as is the domain element. A difference from pSather is that in C*, the partitioning and partition locations of domain array are determined at compile-time, whereas in pSather, the chunk locations are determined during program execution.
- The size of a domain array is fixed at declaration, and therefore there is no way to dynamically change its size, even when the number of needed domain elements changes.
- Because the idea of domain is closely associated with the data-parallel construct (domain select statement), it is not obvious that a domain object can exist independent of a domain array.

In PC++ [46], there is the concept of a homogeneous collection of objects which is analogous to the domain array in C*. The data-parallel construct is also similar to that in C*. One major difference is that a collection class in PC++ only needs to know the interface of the type of its elements. Thus any element type which satisfies the interface can be used in the collection. This allows the code in a collection to be reused; there is no similar facility for code reuse in C*.

There is also a research effort on dpSather [64] to add only "loosely synchronous data parallelism" to sequential Sather, without the notions of threads, synchronization etc. dpSather has bulk types; the elements in a bulk type is finer-grained than chunks. A variable can be declared to have a bulk type by writing "x:**par** <class>"; an invocation "`x.f`" then calls `f` in parallel on all elements of the bulk data. The "loosely synchronous data parallelism" breaks away from SIMD execution and is similar to our approach.

The ideas behind the class `SPREAD{T}` were motivated by replicated objects as in [57], concurrent aggregates [16] and the spread arrays in Split-C [21].

9 Future Research

This report doesn't solve all problems that occur when going from Sather to pSather. We see the following main directions for further research:

9.1 Memory Management [by David Stoutamire]

9.1.1 Allocation

PSather requires maintenance of local heaps on each cluster as well as distributed management of global replicated space (in which each heap object resides at the same position in each local address space). [20] Replicated allocation requires synchronization between threads and the garbage collector and will be part of the same runtime microkernel that handles thread scheduling, messages and garbage collection.

An obvious way to manage replicated memory is dedicate a single cluster to service all requests for replicated heap allocation. However, this could become a bottleneck at large scales. Alternatively, each cluster could exclusively manage a portion of the address space, passing on requests to other clusters when insufficient free space remains. This would not have the bottleneck but would waste increasing space as the scale increases due to fragmentation (replicated objects could not be allocated if they would span that portion of the space managed by a single cluster). These two approaches can be balanced by having some subset of the clusters manage exclusive regions, as determined by consideration of the architecture at hand.

9.1.2 Garbage Collection

There is a large literature on distributed garbage collection techniques [1]. Current versions of pSather use uniprocessor conservative garbage collection [13] and avoid collection by conservatively identifying known garbage at compile time [48]). There are a number of constraints that the pSather garbage collector has to meet:

- To the extent possible, processors must refrain from stopping other processors from doing useful work while garbage collecting. Performance of a program may rely on short remote calls finishing quickly, so it may be necessary for the garbage collector to get out of the way by being incremental.

- Distributed machines often have less available memory at each cluster than their single processor counterparts. Therefore, for some applications efficiency of memory use is important; the management of memory should avoid excessive fragmentation or space wasted due to multiple semispaces.

- Locality should be exploited. To a large extent objects will be referred to only by other local objects. This should be exploited by allowing garbage collection to proceed at each cluster without the attention of other clusters. This indicates the need to be able to identify objects that may be referred to remotely.

- Compiler knowledge of the structure and nature of object types should be used to minimize collection effort. For instance, in some cases the compiler can identify types of objects or particular objects which cannot become remotely referenced or form cyclic structures; this may be used to avoid some work.

[20] On machines where direct control over address space placement is not available, other schemes using indirection are possible. However, this would negate the performance advantage of using `$SPREAD`.

9.1.3 Proposed method of garbage collection

There are many possible choices to be made about such a distributed garbage collector. Here is one possible implementation for the CM-5, a machine without hardware support for the shared memory abstraction:

Memory allocation occurs in the runtime microkernel as a result of a call by a thread requesting memory. If insufficient memory exists, then a local garbage collection occurs. If there is still insufficient memory, a global collection occurs. Objects are not relocated once allocated; this eliminates many possible synchronization points between processors that may otherwise have to occur to handle forwarding pointers, and allows the use of the `destroy` operation. Local collection is a combination of incremental methods (to the extent that the architecture allows this to be done efficiently) and mark and sweep.

To allow local collections, each processor keeps a record of all objects which might be externally referenced, and treats these as roots for each local collection. The table is added to during remote calls and remote writes (all referenced items placed in a message must be inserted) as well as when objects are placed in `PACKET` objects. Note that this table is conservative; entries can only be removed after a global garbage collection identifies they are no longer needed.

The best way to avoid garbage collection is to avoid generating garbage. The current pSather compiler attempts to do this by flow analysis, and by placing provably short lived objects on the stack. Sather provides the `destroy` operation to allow explict invalidation (or direct freeing if runtime checks are off) of objects as well.

The compiler may be able to identify some types as impossible to form remote references to. Such types can be dealt with by reference counting; a word or fraction of a word can be used to count the number of references. When this goes to zero, the object may be put on a list to be reclaimed and all recursive references have their counts adjusted. This may work especially well when the compiler can determine that the type cannot form cyclic structures, for example by noting the absence of cycles in the type usage graph. The advantage to this technique is that it can be made real-time; the garbage collector can operate in bounded time in response to a need to execute a new thread or attribute access. On the negative side this requires extra operations on every reference assignment, including memory system access to the object in question. Whether this is worthwhile may depend on the number of objects that can be reclaimed in this way and their manipulation, and is likely program dependent.

A distributed variant of this that does not require message passing is to keep single-bit saturating reference counts in the references to allow reclamation of objects that do not become aliased. In this technique, a newly created object's reference has a cleared bit which is set in both lhs and rhs on a reference assignment; overwriting a reference with a clear bit allows the object to be reclaimed. Again, this will require some overhead on each reference assignment, but much less than for full reference counting, and with better memory system behavior.

Other techniques will be required because the above techniques may not reclaim data when it is cyclic, remote, or aliased. Because objects are not relocated, a mark and sweep will suffice for local collection when recovery of reference counted structures fails to provide memory. An incremental version can be implemented efficiently when hardware memory management is available [12]. If such hardware is not present, then a compiler-emitted read or write barrier (card marking) could also be used to be incremental. Whether this is necessary depends on the frequency and tolerable latency of local garbage collections, which again may be program dependent. Remote attribute accesses (read and writes) may proceed concurrently with local garbage collection, as long as exiting references are properly added to the list of remotely referenced objects. Remote writes are not a problem because any new references created by a write must already have existed in the table of local objects with remote references.

When the local collect fails, a global garbage collect must be undertaken to remove spurious entries in the entrance tables; this is probably also the best point to reclaim replicated heap objects.

This can occur by stopping the program and doing a concurrent mark and sweep, or by doing an incremental version relying on hardware memory management, or again by barriers emitted by the compiler. Global collection is complicated by the possibility that references may exist in messages still in transit in the network; it is likely that all nodes must at some point be prevented from sending messages long enough to guarantee conservatism.

9.2 Construction of Parallel Class Libraries

pSather was designed under the assumption that there shouldn't be one single layer of abstraction that separates the user from the machine. Rather we envision a layered library of classes with distributed data structures and parallel algorithms. This library should be open on different levels such that it pleases the programmer who is willing to invest some effort to obtain utmost performance as well as the scientist looking for an immediate solution to his problem. In addition to that, the library should be structured such that it relies on a limited number of architecture-dependent classes. One may argue that this is a general problem in the design of class libraries. We think, however, that this is a particularly difficult task due to the broad architectural diversity of parallel computers. pSather offers a powerful set of building blocks for such libraries, but the structure of the libraries is subject to further research.

9.3 Atomicity and Consistency of Memory Operations

The goal of all weak memory consistency approaches is to guarantee sequential consistency for "properly synchronized" programs. In section 3.5.2, we very briefly introduced the idea of weak consistency and consistency ensuring operations on the language level. It seems that this is one of the first attempts to define consistency on the language level and work needs to be done to formally define it and relate it to hardware-based consistency models. It is particularly important to show that this model is a weaker form of the well known weak consistency. This would allow us to put the new model into the hierarchy of existing consistency models. Thus, all architectures supporting stronger consistency models would automatically satisfy our model. Furthermore, it should be interesting to clearly characterize "properly synchronized" programs. Can the compiler in common cases detect such programs and issue warnings? Another question is the worst case behavior of our model in the case of "badly synchronized" programs. What are the minimum safety guarantees we can give in this case? Would a system help that could generate code with higher consistency guarantees for debugging? How is the performance of the new consistency model compared with stronger consistency models? Are there implementations that can take advantage of our weak consistency definition?

9.4 Improving the Efficiency of Gates

Gates (see section 3.3) are very powerful and versatile synchronization constructs. Most of the examples in this report would be more complex and error-prone if only traditional primitives were available. However, from our experience so far, it is often the case that only part of the gate functionality is used by a statement. This may, depending on the implementation, incur unnecessary overhead both in space and time for the unused functionality. One possible solution would be to split the gate class into a hierarchy of multiple built-in classes with gate as the most sophisticated synchronization primitive. Possible classes include locks only dealing with the lock status or futures only dealing with the binding status of gates. Another approach would be that the compiler automatically generates reduced code for a gate where only part of the functionality is used. It remains open to what extent this is automatically possible.

9.5 Extending Other Programming Languages Like pSather

The layered model of extensions from Sather to pSather is applicable to other modern object-oriented programming languages, too. We should investigate to what extent features of different existing programming languages help or inhibit building pSather-like extensions on top.

9.6 pSather on LAN-coupled Workstation Clusters

There is a great deal of current interest in using workstation clusters as a single distributed-memory computing system. It is tempting to extend the idea of pSather to such systems. The problem is that the shared address space as a low-level language abstraction requires a low-latency network. If well implemented on multiprocessors, round-trip latencies for memory accesses to remote clusters cost about two orders of magnitude more than a local cache-hit. Minimum round-trip latencies over a LAN in current realizations are still at least two orders of magnitude slower. There are proposed research programs to greatly reduce LAN latencies and success here would allow pSather to work well on such systems. The CM-5 implementation uses only processor-local techniques and (active) messages and should port without great difficulty to a LAN. But the fastest memory access times continue to improve and the nanosecond/foot limitation suggests that peak performance will continue to require tightly packaged systems.

9.7 Instrumentation and Debugging [by Mark Minas]

When programming in parallel, concurrency, nondeterminism, and synchronization add complexity that makes debugging even harder than debugging sequential programs. This is particularly true for pSather programs. Experiences with programming applications in pSather have shown the need of debugging tools.

Errors in parallel programs can be classified in two distinct categories: Performance errors like bottlenecks and insufficient load balancing reduce the program performance without changing program results, while the second class of errors includes errors leading to wrong or unpredictable program behavior. Conventional errors occurring in sequential programs belong to this class, as well as race conditions and deadlocks.

The obvious way to detect performance errors is visualizing the execution of a program including its communication behavior, its frequency of memory accesses, and so on. The question of which are the program and execution parameters that should be visualized and how the information should be presented in order to effectively and efficiently detect performance errors [44] is an open research issue. For the second error class no uniform debugging technique exists. Much effort is spent on the solution of this problem. But so far, no completely satisfying solution has been found.

We plan to construct a debugging tool for pSather that mainly addresses errors of the second class by using the classic technique of *cyclical debugging* [54] that has proved useful for sequential programs: The sequential program is repeatedly stopped during execution, the program state examined, and the program either continued or reexecuted in order to stop at an earlier point in execution to locate an error. This works for sequential programs since they consist of only one thread of control and their behavior is deterministic. Thus the program behaves identically in each execution and reexecution. In general, this is not true for pSather programs. Because of the inherent nondeterminism, the program may behave differently when reexecuted. The standard solution for this problem is *instant replay* [43]: The generated code is automatically instrumented such that it writes traces of its behavior during its execution. For pSather these traces must include the sequence of shared memory accesses, timing of synchronizations, etc. After termination of the program these traces are used by the program to replay the first execution, thus leading to an identical behavior. The replaying program must have been further instrumented to obey the traces.

Instrumenting leads to a dilemma: On the one hand, the instrumentation should generate sufficient trace information that allows an identical replay of the original execution. On the other hand, instrumenting the code alters the program behavior ("*Probe Effect*" [45], "*Heisenberg Uncertainty*" [29].) In order to minimize such changes, instrumenting should be reduced. Instant replay is a helpful method by tracing only causalities between events rather than transmitted data, thus minimizing the amount of information to be traced. Data can be reconstructed during replay. Furthermore, writing traces results in a large amount of data increasing monotonically with time. Due to space limitations this amount of data must be restricted. A possible solution to this problem might be introducing checkpoints to the program and removing traces gathered so far whenever such a checkpoint is reached during execution.

We plan to use the outlined monitoring and replay approach as a platform for several debugging methods:

- Since the traces contain all the information that is needed for an identical reexecution, the replayed program need not to be executed on the same machine as during the monitoring phase. The program can be replayed on a workstation even if it has been monitored on a large parallel machine.

- Many debuggers for sequential programs allow checking of predicates while executing the program and stopping the program as soon as one predicate is satisfied. Thus, predicates generalize the idea of breakpoints. Similarly, we plan to insert predicates to program replay control. One difficulty that must be dealt with is the absence of directly observable global states in distributed systems, which must be addressed for pSather programs. Some work in this direction has been done in [53, 56]. Future work will investigate which classes of predicates are needed. A sort of temporal logic is a possible candidate, since many problems when programming in parallel are timing problems.

- With the ability to insert arbitrary breakpoints by checking predicates the monitoring and replay approach allows cyclical debugging. It is planned to use standard techniques to investigate program states of stopped pSather programs.

- Further instrumentation of the program is possible without affecting the program behavior. Gathering this information while replaying allows for visualization needed especially to detect performance errors.

- So far, the traces are used as an aid for reproducibility of program behavior. Recent research on debugging parallel programs deals with analyzing the traces by itself to detect race conditions and candidates for deadlocks [10, 36]. Using such techniques for traces of pSather programs may lead to the detection of errors that cannot be found by simply replaying the program execution.

10 Conclusions

The problem of general purpose parallel programming remains one of the most challenging and important research tasks. Constructing flexible and efficient parallel data structures and algorithms will remain difficult at least for some time. The premise behind this paper is that appropriate language constructs can both help in the development of good parallel abstractions and in their utilization. The pSather approach relies on the underlying clarity, safety, and efficiency of the Sather language and libraries. The additional `GATE` class and associated constructs provide a general mechanism for directly coordinating multiple execution threads. The **`with-near`**-statement and copying mechanisms provide additional support for mapping the pSather shared address space model to distributed memory architectures. The additional classes `SPREAD{T}` and `$DIST{T}` and the **`dist`**-statement simplify the use of distributed data structures in general computation. A preliminary version of these mechanism has been built on the CM-5 and is yielding promising initial results. Now we need to construct the full system described here and try it on a wide range of problems. We hope to release a preliminary version of pSather for experimental use early in 1994.

11 Acknowledgements

Sather 1.0 is the basis on which pSather relies. We would therefore like to thank all who have contributed to the design of Sather 1.0, in particular Steve Omohundro, the main designer of Sather and the Sather working group (including David Bailey, Joachim Beer, Ben Gomes and David Stoutamire at ICSI, and Heinz Schmidt from CSIRO). We even borrow from the yet unpublished Sather 1.0 language definition for the section about serial Sather. Furthermore, we would like to thank Franco Mazzanti who was instrumental in the design of monitors (now renamed as gates) and the first implementation of pSather on the Sequent Symmetry. In addition, we owe thanks to Jeff Bilmes, Thomas Rauber, and Hans Rohnert who have contributed to the design and implementation of pSather. Last but not least we would like to thank Clemens Szyperski for carefully reading preliminary versions of this paper and for pointing out the conceptual deficiencies in earlier designs. Special thanks go to David Stoutamire (Section 9.1 and Mark Minas (Section 9.7 for their contributions to the future work section.

A Complete Grammar of pSather 1.0

In this appendix we give the full grammar of pSather 1.0 and point out the new or changed productions compared to Sather 1.0 by putting them into boxes.

A.1 Declarations

class_def_list ⇒ [class_def] | class_def_list ; [class_def]

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

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

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)*

class_elt_list ⇒ [class_elt] | class_elt_list ; [class_elt]

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

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

ident_list ⇒ ident (, ident)*

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

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

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

arg_dec ⇒ [ident_list :] type_spec

stmt_list ⇒ [stmt] | stmt_list ; [stmt]

iter_def ⇒ [private] iter 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 [!] (, ident [!])*:] type_spec

include_clause include type_spec :: feat_mod | [private] include type_spec [feat_mod (, feat_mod
)*]

feat_mod ⇒ ident [(type_spec [!] (, type_spec [!]) *)][: type_spec] ⇒ [[private | readonly] ident]

A.2 Statements

stmt ⇒ *dec_stmt* | *simple_assign_stmt* | *tuple_assign_stmt* | *if_stmt* | *loop_stmt* | *return_stmt* |
yield_stmt | *quit_stmt* | *case_stmt* | *typecase_stmt* | *assert_stmt* | *protect_stmt* | *raise_stmt* |
expr_stmt | *cobegin_stmt* | *lock_stmt* | *try_stmt* | *unlock_stmt* | *near_stmt* | *dist_stmt* | *sync_stmt*

dec_stmt ⇒ *ident_list* `:` *type_spec*

simple_assign_stmt ⇒ *lhs_elt* `:-` | `:=` *expr*

lhs_elt ⇒ *expr* | `_` | *ident* `:` [*type_spec*]

tuple_assign_stmt ⇒ `#(` *lhs_elt* (`,` *lhs_elt*)*`)`) `:=` *expr* | *lhs_elt* `:=` `#(` *rhs_elt* (`,` *rhs_elt*)*`)`)

rhs_elt ⇒ *expr* | `_`

if_stmt ⇒ `if` *expr* `then` *stmt_list*
(`elsif` *expr* `then` *stmt_list*)*
[`else` *stmt_list*] `end`

loop_stmt ⇒ `loop` *stmt_list* `end`

return_stmt ⇒ `return`

yield_stmt ⇒ `yield`

quit_stmt ⇒ `quit`

case_stmt ⇒ `case` *expr* (`when` *expr* (`,` *expr*)* `then` *stmt_list*)*
[`else` *stmt_list*] `end`

typecase_stmt ⇒ `typecase` *ident* [`:` [*type_spec*] `:=` *expr*]
(`when` *type_spec_list* `then` *stmt_list*)*[`else` *stmt_list*] `end`

assert_stmt ⇒ `assert` *expr*

protect_stmt ⇒ `protect` *stmt_list* (`against` *type_spec_list* `then` *stmt_list*)* `end`

raise_stmt ⇒ `raise`

expr_stmt ⇒ [`:-`] *expr*

cobegin_stmt ⇒ `cobegin` *stmt_list* `end`

lock_stmt ⇒ `lock` *expr_list* `then` *stmt_list* `end`

try_stmt ⇒ `try` *expr_list* `then` *stmt_list* [`else` *stmt_list*] `end`

unlock_stmt ⇒ `unlock` *expr*

near_stmt ⇒ `with` *ident_list* `near` *stmt_list* [`else` *stmt_list*] `end`

dist_stmt ⇒ `dist` *expr* `as` *ident* (`,` *expr* `as` *ident*)*`do` *stmt_list* `end` *sync_stmt* ⇒ `sync`

A.3 Expressions

expr ⇒ *local_expr* | *call_expr* | *void_expr* | *cons_expr* | *bound_cons_expr* | *sugar_expr* | *and_expr* | *or_expr* | *not_expr* | *equal_expr* | *initial_expr* | *bool_lit_expr* | *char_lit_expr* | *str_lit_expr* | *int_lit_expr* | *flt_lit_expr*

local_expr ⇒ *ident*

call_expr ⇒ *[expr* `.` | *type_spec* `::`*] ident [*`(` *expr_list* `)`*] [*`@` *expr]*

expr_list ⇒ *expr (*`,` *expr)**

void_expr ⇒ `void`

cons_expr ⇒ `#` *[type_spec] [*`(` *cons_elt (*`,` *cons_elt)**`)` *] [* `@` *expr]*

cons_elt ⇒ | *[ident* `:=`*] (expr* | `#` *digit+)*

bound_cons_expr ⇒ *(*`#ROUT` | `#ITER`*)*`(` *[type_spec* `::` | *bound_arg* `.`*] ident [*`(`*bound_arg (*`,` *bound_arg)**`)`*][*`:`*type_spec]* `)`

bound_arg ⇒ *expr* | `_`*[*`:` *type_spec]*

sugar_expr ⇒ *expr binary_op expr* | `-` *expr* | *[expr]* `[` *expr_list* `]` | `(` *expr* `)`

binary_op ⇒ `+` | `-` | `*` | `/` | `^` | `%` | `&` | `/=` | `<` | `<=` | `>` | `>=`

and_expr ⇒ *expr* `and` *expr*

or_expr ⇒ *expr* `or` *expr*

not_expr ⇒ `not` *expr*

equal_expr ⇒ *expr* `=` *expr*

initial_expr ⇒ `initial(` *expr* `)`

A.4 Lexical Elements in pSather

ident ⇒ *letter (letter* | *digit* | `_`*)**

class_name ⇒ *[*`$`*] uppercase_letter (uppercase_letter* | *digit* | `_`*)**

letter ⇒ *lowercase_letter* | *uppercase_letter*

lowercase_letter ⇒ `a` | `b` | `c` | `d` | `e` | `f` | `g` | `h` | `i` | `j` | `k` | `l` | `m` | `n` | `o` | `p` | `q` | `r` | `s` | `t` | `u` | `v` | `w` | `x` | `y` | `z` |

uppercase_letter ⇒ `A` | `B` | `C` | `D` | `E` | `F` | `G` | `H` | `I` | `J` | `K` | `L` | `M` | `N` | `O` | `P` | `Q` | `R` | `S` | `T` | `U` | `V` | `W` | `X` | `Y` | `Z` *digit* ⇒ `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

special_symbol ⇒ `(` | `)` | `[` | `]` | `{` | `}` | `,` | `.` | `;` | `:` | `$` | `_` | `+` | `-` | `*` | `/` | `=` | `<` | `>` | `#` | `^` | `%` | `&` | `/=` | `<=` | `>=` | `:=` | `::` | `:-` | `@`

A.5 Predefined Identifiers

special_classnames ⇒ `$EXCEPTION` | `$EXTOB` | `$OB` | `ARRAY` | `BITS` | `BOOL` | `CHAR` | `FLT` | `FLTD` | `FLTE` | `FLTDE` | `INT` | `INTFIX` | `INTINF` | `$REHASH` | `SAME` | `STR` | `SYS` | `TYPE` | `$GATE` | `GATE` | `$GATE0` | `GATE0` | `$SPREAD` | `SPREAD` | `$DIST` | `DIST` |

special_featurenames ⇒ `aget` | `append` | `arg` | `aset` | `break` | `clear` | `copy` | `enqueue` | `exception` | `fix` | `id` | `destroy` | `is_geq` | `is_gt` | `is_leq` | `is_lt` | | `main` | `minus` | `mod` | `negate` | `over` | `plus` | `pow` | `read` | `res` | `self` | `set` | `str` | `take` | `times` | `type` | `until` | `void` | `while`

A.6 Literals

bool_lit_expr ⇒ `true` | `false`

char_lit_expr ⇒ `'` *(ISO_character* | `\` *escape_seq)* `'`

escape_seq ⇒ `a` | `b` | `f` | `n` | `r` | `t` | `v` | `\` | `'` | `"` | *octal_digit+*

str_lit_expr ⇒ *(*`"` *ISO_character** `"`*)+*

int_lit_expr ⇒ *binary_int* | *octal_int* | *decimal_int* | *hex_int*

binary_int ⇒ `0b` *binary_digit+*

binary_digit ⇒ `0` | `1`

octal_int ⇒ `0o` *octal_digit+*

octal_digit ⇒ `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7`

decimal_int ⇒ *digit+*

hex_digit ⇒ *digit* | `a` | `b` | `c` | `d` | `e` | `f`

hex_int ⇒ `0x` *hex_digit+*

flt_lit_expr ⇒ *digit+* `.` *digit+ [*`e` *[*`+` | `-`*] digit+]* | `NaN` | `Inf`

References

[1] S. Abdullahi, E. Miranda, and G. Ringwood. Collection schemes for distributed garbage. In *Proceedings of the International Workshop on Memory Management (IWMM 92) - LNCS 637*, pages 43–81. Springer-Verlag, September 1992.

[2] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. The MIT Press, Cambridge Massachusetts, 1986.

[3] Eugene Albert, Joan D. Lukas, and Guy L. Steele Jr. Data Parallel Computers and the FORALL Statement. *Journal of Parallel and Distributed Computing*, 13:185–192, 1991.

[4] Pierre America. *Issues in the Design of a Parallel Object-Oriented Language*. Philips Research Laboratories, Eindhoven and University of Amsterdam, March 1 1989. Part of POOL2/PTC Distribution Package.

[5] Gregory R. Andrews and Fred B. Schneider. Concepts and Notations for Concurrent Programming. *ACM Computing Surveys*, 15(1):3–43, March 1983.

[6] Henri E. Bal, Andrew S. Tanenbaum, and M. Frans Kaashoek. Orca: A language for distributed programming. *SIGPLAN Notices*, 25(5):17–24, 1990.

[7] J. G. P. Barnes. An Overview of Ada. *Software – Practice and Experience*, 10(11):851–887, 1980.

[8] Paul S. Barth, Rishiyur S. Nikhil, and Arvind. M-Structures: Extending a Parallel, Non-Strict, Functional Language with State. In *Functional Programming Languages and Computer Architecture, 5th ACM Conference Proceedings, Lecture Notes in Computer Science 523*, pages 538–568, August 1991.

[9] John K. Bennett. The Design and Implementation of Distributed Smalltalk. In *OOPSLA '87 Conference Proceedings*, pages 318–330, October 4-8 1987. Proceedings also published as: SIGPLAN Notices, Vol 22, No 12, December 1987.

[10] Anton Beranek. Data race detection based on execution replay for parallel applications. In *Proceedings of CONPAR '92*, pages 109–114, Lyon, France, September 1992.

[11] Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. PRESTO: A System for Object-Oriented Parallel Programming. *Software – Practice and Experience*, 18(8):713–732, August 1988.

[12] H. Boehm, A. Demers, and S. Shenker. Mostly parallel garbage collection. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 157–164. ACM, June 1991. Also available as: SIGPLAN Notices Vol 26, No. 6, June 1991.

[13] H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, pages 807–820, September 1988.

[14] Peter A. Buhr and Richard A. Stroobosscher. *μC++ Annotated Reference Manual Version 3.4.4*. University of Waterloo, August 18 1992. Part of μC++ Distribution Package.

[15] Rohit Chandra, Anoop Gupta, and John L. Hennessy. Integrating Concurrency and Data Abstraction in a Parallel Programming Language. Technical Report CSL-TR-92-511, Computer Systems Laboratory, Stanford University, February 1992.

[16] Andrew Andai Chien. *Concurrent Aggregates (CA): An Object-Oriented Language for Fine-Grained Message-Passing Machines.* PhD thesis, MIT, July 1990. Also available as: MIT Artificial Intelligence Laboratory, Technical Report 1248.

[17] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT electrical engineering and computer science series. MIT Press, Cambridge, Massachusetts, 1990.

[18] Thinking Machines Corporation. *The Connection Machine CM-5 Technical Summary.* Thinking Machines Corporation, Cambridge Massachusetts, October 1991.

[19] P. J. Courtois, F. Heymans, and David L. Parnas. Concurrent control with readers and writers. *Communications of the ACM*, 10:667–668, October 1971.

[20] Lawrence A. Crowl. *Architectural adaptability in parallel programming.* PhD thesis, University of Rochester, May 1991.

[21] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in split-c. In *to be published in Proceedings of Supercomputing 93*, 1993.

[22] Edsger W. Dijkstra. Co-operating sequential processes. In F. Genuys, editor, *Programming Languages.* Academic Press, 1965.

[23] Michel Dubois, Christoph Scheurich, and Fayé Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual Symposium on Computer Architecture*, pages 434–442, June 1986.

[24] Akinori Yonezawa (ed). *ABCL – An object-oriented concurrent system.* MIT Press, Cambridge, Massachusetts, 1990.

[25] Michel Cekleov et al. SPARCCenter2000: Multiprocessing for the 90's! In *COMPCON spring 1993.* IEEE COmputer Society Press, February 1993.

[26] Jerome A. Feldman. High level programming for distributed computing. *Communications of the ACM*, 22(6):353–368, June 1979.

[27] 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., 1991.

[28] Jerome A. Feldman, Chu-Cheow Lim, and Thomas Rauber. The Shared-Memory Langauge pSather on a Distributed-Memory Multiprocessor. In *Second Workshop on Languages, Compilers, and Run-Time Environments for Distributed-Memory Multiprocessors*, Boulder, Colorado, Sept 30 - Oct 2 1992.

[29] Jason Gait. A debugger for concurrent programs. *Software - Practice and Experience*, 15(6):539–554, June 1985.

[30] Dennis Gannon, Jenq Kuen Lee, and Srinivas Narayana. On Using Object Oriented Parallel Programming to Build Distributed Algebraic Abstractions. In *Proceedings of CONPAR 92 - LNCS 634.* Springer-Verlag, September 1992.

[31] Kouroush Gharachorloo, Anoop Gupta, and John Hennessy. Performance evaluation of memory consistency models for shared memory multiprocessors. In *Proceedings of the 18th Annual Symposium on Computer Architecture*, pages 245–257, June 1991.

[32] Kouroush Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared memory multiprocessors. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 15–26, June 1990.

[33] Per Brinch Hansen. The programming language concurrent pascal. *IEEE Transactions on Software Engineering*, SE-1:199–207, June 1975.

[34] Per Brinch Hansen. Monitors and Concurrent Pascal: A Personal History. In *Second ACM SIGPLAN History of Programming Languages Conference (HOPL-II)*, Cambridge, Massachusetts, USA, April 20-23 1993.

[35] Philip J. Hatcher, Anthony J. Lapadula, Robert R. Jones, Michael J. Quinn, and Ray J. Anderson. A Production-Quality C* Compiler for a Hypercube Multicomputer. Technical Report TR 90-80-3, Oregon State University, Computer Science Department, 1990.

[36] David P. Helmbold, Charles E. McDowell, and Jian-Zhong Wang. Determining possible event orders by analyzing sequential traces. *IEEE Transactions on Parallel and Distributed Systems*, 4(7):827–839, July 1993.

[37] John Hennessy and David Patterson. *Computer Architecture, a Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.

[38] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17:549–557, October 1974.

[39] C. A. R. Hoare. *Communicating Sequential Processes*, pages 136–148. IEEE Computer Society Press, 1989. Also published in Communications of the ACM, August 1978, pp 666-677.

[40] Eric Jul et al. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[41] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):241–248, September 1979.

[42] Butler W. Lampson and David D. Redell. Experience with Processes and Monitors in Mesa. *Communications of the ACM*, 23(2):105–117, February 1980.

[43] Thomas J. LeBlanc and John M. Mellor-Crummey. Debugging parallel programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[44] Thomas J. LeBlanc, John M. Mellor-Crummey, and Robert J. Fowler. Analyzing parallel program executions using multiple views. *Journal of Parallel and Distributed Computing*, 9:203–217, June 1990.

[45] Carol H. LeDoux and D. Stott Parker, Jr. Saving traces for Ada debugging. *Ada in Use, Proceedings of the Ada International Conference, published in ACM Ada Letters*, 5(2):97–108, September 1985.

[46] Jenq Kuen Lee and Dennis Gannon. Object Oriented Parallel Programming Experiments and Results. In *Proceedings of Supercomputing '91*. IEEE Computer Society Press and ACM SIGARCH, November 1991.

[47] Daniel Lenoski, James Laudon, et al. The Stanford Dash Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[48] Chu-Cheow Lim. *A Parallel Object-Oriented System for Realizing Reusable and Efficient Data Abstractions*. PhD thesis, University of California at Berkeley, 1993. also available as technical report TR-93-063, International Computer Science Institute.

[49] Chu-Cheow Lim, Jerome A. Feldman, and Stephan Murer. Unifying control- and data-parallelism in an object-oriented language. In *Proceedings of the 1993 Joint Symposium on Parallel Processing*, Tokyo, Japan, May 17 - May 19 1993.

[50] Tom Lovett and Shreekant Thakkar. The Symmetry Multiprocessor System. In *Proceedings of International Conference on Parallel Processing*, pages 303–310. Pennsylvania State University Press, University Park, PA., 1988.

[51] Steven E. Lucco. Parallel Programming in a Virtual Object Space. In *Proceedings OOPSLA '87*, pages 26–34. ACM, December 1987. Also available as: SIGPLAN Notice Vol 22, No. 12, Dec 1987.

[52] Steven E. Lucco and David P. Anderson. Tarmac: a language system substrate based on mobile memory. Technical Report UCB/CSD 89/525, Computer Science Division (EECS), University of California, Berkeley, CA 94720, November 1989.

[53] Yoshifumi Manabe and Makoto Imase. Global conditions in debugging distributed programs. *Journal of Parallel and Distributed Computing*, 15:62–69, 1992.

[54] Charles E. McDowell and David P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–622, December 1989.

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

[56] Barton P. Miller and Jong-Deok Choi. Breakpoints and halting in distributed programs. In *Proceedings of the Eighth International Conference on Distributed Computing Systems*, pages 316–323, 1988.

[57] Stephan Murer and Philipp Färber. A scalable distributed shared memory system. In *Proceedings of CONPAR'92-VAPP V - LNCS 634*, Lyon, France, September 1992. Springer-Verlag.

[58] Stephan Murer, Stephen Omohundro, and Clemens Szyperski. Sather iters: Object-oriented iteration abstraction. Technical Report TR-93-045, International Computer Science Institute, Berkeley, Ca., 1993.

[59] O. M. Nierstrasz. Active Objects in Hybrid. In *OOPSLA 1987 Conference Proceedings*, pages 243–253, Oct 4 - Oct 8 1987.

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

[61] Stephen M. Omohundro. The Sather 1.0 Specification. Technical report, International Computer Science Institute, Berkeley, Ca., 1992 (in preparation).

[62] Dick Pountain. A tutorial introduction to OCCAM programming. Inmos Occam Manual.

[63] Jr R. H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[64] H. W. Schmidt. Data-Parallel Object-Oriented Programming. In *Proc. of the Fifth Australian Supercomputer Conference, Melbourne (AUS): Univ. Melbourne*, December 1992.

[65] Reference Manual for the Ada Programming Language. United States Department of Defense, July 1982.

[66] Peter Wegner and Scott A. Smolka. Processes, Tasks, and Monitors: A Comparative Study of Concurrent Programming Primitives. *IEEE Transactions on Software Engineering*, SE-9(4):446–462, July 1983.

[67] Y. Yokote and M. Tokoro. Experience and evolution of Concurrent Smalltalk. In *Proceedings of OOPSLA*, pages 406–415, Orlando, Florida, December 1987. ACM.