[3] L. Cardelli: "A Semantics of Multiple Inheritance" in *Kahn et al (eds): Semantics of Data Types,* New York: Springer, LNCS 173, pp. 51–67, 1984

[4] L. Cardelli: "Basic polymorphic type checking", *Science of Computer Programming* **8** pp. 147–172 (1987)

[5] C. Chambers, D. Ungar and E. Lee: "An efficient implementation of SELF, a dymically-typed object-oriented language based on prototypes." *OOPSLA 89 Conf. Proc., Special Issue, SIGPLAN NOTICES,* **24**(10), 1989

[6] C.C. Lim and A. Stolcke: *Sather Language Design and Performance Evaluation.* TR-91-034, ICSI, 1991

[7] T. Coquand, C. Gunter, G. Winskel: "Domain Theoretic Models of Polymorphism." in *Information and Computation* **81**(2), pp. 123–167, 1989

[8] W.R. Cook: "A Proposal for Making Eiffel Type-safe." in S. Cook (Ed.): ACM ECOOP 89 Conf. Proc., British Computer Society Workshop Series, pp. 57–70, 1989

[9] Interactive Software Engineering: *Eiffel The Language, Ref. Manual,* ISE, TR-EI-17/RM, 1989

[10] J.A. Feldman, C.C. Lim, F. Mazzanti: *pSather Monitors: Design, Tutorial, Rationale and Implementation. ICSI, TR-91-031* 1991

[11] U. Hoelzle, C. Chambers, D. Ungar: "Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches", in O. Nierstrasz (ed.): *Proc. ECOOP 91,* 1991

[12] A.K. Jones and B.H. Liskov, "A Language Extension for Controlling Access to Shared Data." *IEEE Trans. Soft. Eng.* **2**(4), pp. 277–285 (1976)

[13] S. Keene, *Object-oriented Programming in Common Lisp.* Addison-Wesley, 1989

[14] S. Keene, "Multiple Inheritance in CLOS". *JOOP* **2**(5), pp. 75–77 (1990)

[15] G. Kiczales, J. des Rivieres and D.G. Bobrow: *The Art of the Metaobject Protocol.* MIT Press, 1991

[16] B. Krämer and H.W. Schmidt: "Architecture and Functionality of a Specification Environment for Distributed Software". *IEEE Conf. Proc. COMPSAC 90,* pp. 617–622, 1990

[17] B. Meyer, *Object-oriented Software Construction,* Prentice Hall, 1988

[18] B. Meyer, "Sequential and Concurrent Object-Oriented Programming" *Conf. Proc. Tools '90,*1990

[19] D.A. Moon: "The Common Lisp Object-Oriented Programming Language Standard." in *W. Kim and F. Lochovsky (eds.): Object-Oriented Concepts, Applications, and Databases.* Addison-Wesley, 1988

[20] S.M. Omohundro, *The Sather Language.* ICSI, Berkeley, 1990.

[21] J.R. Rose: "Fast Dispatch Mechanisms for Stock Hardware" *SIGPLAN Notices* **22**(2) pp. 85–94, 1987

[22] H.W. Schmidt, B. Gomes: *ICSIM – An Object-Oriented Connectionist Simulator.* ICSI, Berkeley, TR-91-048, 1991

[23] H.W. Schmidt, J. Bilmes: "Exception Handling in pSather." *Proc. Workshop Exception Handling, ECOOP 91,* Geneva, 1991, to appear.

[24] G. Smolka, W. Nutt, J, A. Goguen and J. Meseguer: "Order-Sorted Equational Computation." in *M. Nivat, H. Aít-Aci (eds.): Resolution of Equations in Algebraic Structures,* Academic Press, 1988

[25] G.L. Steele Jr., *Common Lisp – The Language.* Digital Press, 1990

of some of its implementations make Common Lisp the language of choice in artificial intelligence for many in academia and in industry. The completed ANSI standard adds to the broad acceptability of the language. The Common Lisp Object System with its advanced features, like method combination, dynamic class creation and change, and the meta-object protocol make it ever more attractive for experimental projects in object-orientation.

Eiffel is a young language, the first attempt at a language in which a flexible type system, static typing, inheritance of semantics and other advanced issues were combined and made commercially available. There is a growing interest in the language, particularly in academia and for teaching of high-level language issues. For a young language the suite of tools is acceptable. The language has a few exotic features, partly because common terminology is abandoned in favor of distinctiveness, partly because semantic issues are overstressed. Mostly the language is a success and provides a fresh look at many issues in object-oriented programming. Unfortunately, the language directly competes with C++ because it addresses the same set of users. It appears that in the industrial arena, C++ is likely to be the preferred language for reasons other than elegance and clarity. Perhaps – or hopefully – it will absorb some useful features of Eiffel on the way. There will still be sufficient space for Eiffel to develop further.

Sather is the newest language among the languages compared. It looks much like Eiffel but stresses implementation inheritance, simplicity and efficiency, one reason for static typing in the language. Its semantics differs from Eiffel in a number of respects. Types can be dispatched and non-dispatched. Implementation inheritance is stressed more than semantics inheritance and the order of parent classes defines inheritance like in many other object-oriented languages, avoiding Eiffel's need for low-level conflict resolution. Partly these differences are due to the desire for simplification, partly to allow more pragmatic and higher-level (class-level) forms of code combination and reuse, partly they try to resolve semantical problems of Eiffel. For instance contravariance is required for conformance between redefined and new methods and dynamic parametrization only allows reference to the type of *self*. All this greatly simplifies typing issues and optimization but can make Eiffel programs incompatible with Sather. The restriction to fewer semantic constructs and a few implementation-oriented limiting decisions are made up for by a simpler language semantics that can make programs clearer.

Acknowledgement

Thanks to Danny Bobrow, Pierre Cointee, Richard Gabriel, Nicolas Graube, Gregor Kiszgales and in particular Andreas Paepcke who posed many helpful questions and made several concrete suggestions for improving the text. Explicit thanks also to Jerome Feldman who helped improving the clarity of some paragraphs.

References

[1] D.G. Bobrow, L.G. DeMichiel, R.P. Gabriel, S. Keene, G. Kiczales, and D.A. Moon: *Common Lisp Object System Specification, X3J13 Document 88-002R.* (Also published in *SIGPLAN Notices,* **23,** special issue, Sept. 1988, and in Guy Steele: *Common Lisp, The Language, 2nd ed.,* Digital Press, 1990.)

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

all its descendent constructor types including A. Among the advantages of this distinction are the explicit control of typing, non-dispatched calls, inlining and other optimizations. Semantically, it allows a more precise specification of the intended meaning of a declaration and provides for stricter type checking. A disadvantage is that programmers may choose an undispatched type for efficiency considerations and exclude future unplanned-for extensions that might be possible were subtyping allowed.

Unlike the previous example, the following procedures of class PERSISTENT_STACK show the use of dispatched types and call disambiguation by prefixing the class name.

```
append(s: $STACK{T}) is    -- append s (unmodified) to self.
  i: INT; until i = s.ssize loop push(s[i]); i := i + 1 end
end;

private handle_unknown_type is
  OUT::s("PERSISTENT_STACK {T}: ");    -- std out string
  PERSISTENT::handle_unknown_type;
  ...
end;
```

Disambiguated calls are non-dispatched. *self* is bound to the *void* value when the corresponding routines execute. Given the possibility of disambiguated calls, it makes sense to write non-instantiable classes that comprise only functions whose definitions do not refer to self. In other words disambiguated calls can be used together with function packages containing only non-overloaded, or monomorphic, function packages. Such classes are used to interface to Sather from C packages.

Sather is currently being extended to pSather, an experimental parallel version of the language that adds threads, synchronization, protection and exception handling [10, 23]. By default objects are parallel, i.e., multiple threads can execute in them. A distinguished class called MONITOR combines various efficient low-level synchronization mechanisms such as locking, futures and event-like mechanisms. Higher-level library classes can then define standard or customized synchronization disciplines on top of these MONITOR primitives. The hope is that library classes provide sufficient freedom for trading off parallelism abstractions, mappings to different architectures and efficient, perhaps machine-dependent, implementations. A first experimental implementation of pSather runs on Sequent multi-processors and SPARC.

6 Conclusion

We have given an overview of three object-oriented languages, CLOS, Eiffel and Sather.

Among these Common Lisp builds undoubtedly on the longest experience with related object-oriented dialects. Many useful tools are available, several (full or partial) implementations of CLOS exist on the marketplace, public domain implementations are available and a growing community of CLOS users exists and shares their sources. Compatibility with earlier Lisp dialects is a requirement to the language and therefore many compromises had to be made that make Common Lisp a baroque elephant amongst the various Lisp dialects. Its many existing tools, though, its flexibility in prototyping, its portability and the efficiency

defined in combinations with other classes in the different descendents. Moreover, late checking implies that the body of one routine inherited by different classes may have different type decorations in these different classes. In general this reduces the requirements for compatibility between inherited routines and their redefinitions. For instance, due to late checking, there are no compatibility requirements between an inherited private routine and its redefinition at all.

Sather extends the flexibility of its extensible type system to foreign types. The type F_OB is known to have word size but can be specialized like basic types. It has no predefined operations but can be extended and is restricted to non-dispatched calls. In this way Sather promotes the use of well-tested and professionally maintained foreign packages (currently only C) as *foreign abstract data types* and the compiler can type-check calls to foreign functions.

The capability of subtyping for foreign types makes the foreign interface particularly elegant. For one, no special treatment of foreign types is necessary with respect to typing. More importantly, programmers can take full advantage of type-checking at one of the weakest interfaces, viz., to foreign packages, that are usually hard to debug. An appropriate declaration of foreign functions is thus rewarded in Sather.

The design of Sather was based on Eiffel but focussed more on expressive power, efficiency and simplicity than on the formal and theoretical issues addressed by Eiffel. The language is much smaller than the current Eiffel, it eliminates many keywords and simplifies the syntax and inheritance rules. In Sather, parent classes are listed with features in arbitrary sequence. In conflicts, the last feature wins.

The following code depicts a partial class definition in Sather. The name (PANEL) is followed by a sequence of definitions, among them a reference to an ancestor class (CANVAS) and several attribute definitions.

```
class PANEL is
  CANVAS;
  shared default_title: STR := "No Title"; -- stub initialization.
  constant is_panel: BOOL := true;         -- true here and in subtypes
  private col_gap: INT;                    -- delta pixels between columns. -1 default.
  ...
end;
```

The special type SELF_TYPE is used for dynamic parametrization as in

equal(e: SELF_TYPE*):* BOOL

which implies that a dotted call *x.equal(y)* has type compatible arguments.

Sather allows one to explicitly declare *dispatched types*. If a variable declaration directly specifies a class name as in *a:* FOO then that variable can only hold objects of exactly this type. If the class name is preceded by a dollar sign (indicating a variable type[9]) as in *a:* $FOO then at run time, the variable may hold objects from any descendent class of FOO. Semantically, the constructor type A corresponds to the set of instances carrying the dynamic type A (created with A::*new*). In contrast to this, the dispatched type $A is the union of

[9] and indicating the slightly higher expense of dispatching

ated to that of *layer_create_unit* which may change in future descendents. The rules for such type associations in Eiffel guarantee that the type-checker can use the more refined type information in the descendent context but does not have to reanalyse and/or recompile the current function (*create_component*). It only requires the *signature* (argument and result type) to do its job.

```
layer_create_unit(i: INTEGER): BP_UNIT is
  do Result.create(0) end;
```

The code below contains a few assertions. Pre- and postconditions are specified as *required* and *ensured* assertions, respectively, and play different roles. The caller must guarantee that the assertions at the begin of the definition (require) are satisfied. The implementor must guarantee those at the end (ensure). The advantage of this kind of programming with 'interfaces as contracts' [17] is the separation of responsibilities, in particular, checks need not be built into the 'wrong' program context avoiding the dreaded argument 'overcheck' syndrome of many large modular programs where programs perform the same checks in many places and sometimes often in the same recursive descent.

```
desired_output(i: INTEGER): REAL is
    -- compute the desired output for the i-th output unit
  require 0 <= i; i < output_layer.units.count;
  -- by default we are satisfied
  do Result:=output_layer.units.item(i).output end;
```

Eiffel has many more features that deserve attention and make it a language with scope for the future. Eiffel 3.0, a new version announced by ISE for 1992 addresses several typing issues, such as void object references, optional parameters and more powerful object creation.

5 Sather

Sather combines the (undispatched) procedural style of programming encapsulated abstract types with the object-oriented specialization and recombination of classes. Its typing rules syntactically isolate potential type violations by restricting them to assignments and make Sather *weakly type-safe*, i.e., type-safe under the assumptions that assignments are type-correct. There are well-known techniques, based on explicit type checking, that protect reverse assignments in which (persistent) objects of a more general types are assigned to variables of more specific types. This makes a sublanguage of Sather (with protected reverse assignments) type-safe in the usual sense.

To ease rapid prototyping and provide maximal freedom in reusing existing definitions despite the strong-typing approach, Sather's typing rules are based on *late checking*. This means that inherited definitions need only be type-correct in the descendent context and client calls to actualized parametric classes need only be type-correct in the actualized version of that class. Although this may put more burden on the type-checker, parameterized classes can be used to model open modules (mixins) some of whose operations are only

```
deferred class MULTI_SITE [FROM_OB -> OUTPUT_OBJECT]
  export repeat INPUTS
  inherit INPUTS [ANY_SITE [FROM_OB] ]
  rename output as accumulated_input,
      feedback_error as site_feedback_error
  redefine is_site, as_site, as_unit, output, feedback_error;

  feature
      < class body omitted >
end;
```

wiring single connections rather than putting boxes together as ADT encapsulation intends to promote. The existence of public domain extensions of the Emacs Eiffel mode with commands that produce maximal visibility by collecting the method names in the class body and listing them in the interface is indicative for this.

The reason for this weakness is simultaneously Eiffel's strength. The strictness of Eiffel's interface rules leads to greatly improved robustness and to a clear separation of interface design and implementation. It is a fascinating question whether an interpreter with the 'right' defaults might provide for a graceful degradation of robustness in favor of design freedom in the early development without automatically leading to designs that cannot be turned into robust products later on.

```
create_component(i: INTEGER): BP_LAYER [like layer_create_unit] is
  local j: INTEGER; u: BP_UNIT; k: INTEGER;
  do Result.create(0);
    j := layer_n_units(i);     -- ask descendent: how many units?
    from k := 0 until k = j loop
      u := layer_create_unit(i);     -- make a unit, descendent may help.
      Result.units.push(u);
      k := k + 1
    end;
  end;
```

The *create_component* code presents a function that creates a BP_LAYER[8] with the help of descendent functions called back. The keyword "is" separates function head and body. The function is called *create_component*, takes one argument, an INTEGER, and returns a value of type BP_LAYER, a parameterized type, here actualized with the association type '**like** *layer_create_unit*'. This type expression refers to the result type of *layer_create_unit*, which is supposed to be another function.

layer_create_unit(i) must make a unit of the appropriate type for layer *i* and is likely to be redefined to give the compiler a sufficiently precise result-type in the descendent context. Thus, the *create_component* function is dynamically parameterized. Its result-type is associ-

[8] short for back-propagation layer, a layer in a particular kind of connectionist nets

```
(defmethod connect-input ((self SILENT-INPUT-CONNECTION) (from UNIT) weight)
      (cond ((zerop weight)
             (vector-push-extend from (inactive-in-links self)))
            (t (vector-push-extend from (in-links self))
             (vector-push-extend weight (weights self))
             (connect-output from self))))


(defmethod update :after ((self ACTIVATION-MIXIN) input)
      (declare (ignore input))
      (unless (zerop (delta self))
        (dotimes (i (out-degree self))
          (trigger (out-unit self i) :input-changed))))
```

and can be explained as function composition along the class hierarchy. Method combination has many more facets that we do not have space to describe here. For further details cf. for example [13].

4 Eiffel

A class in Eiffel lists its name, interface, and features. The class below is a *deferred* (non-instantiable) class and the compiler will ignore missing methods. But the interface for all called methods must be present. Features can be declared as *deferred* for that purpose. The class also shows bounded parametrization. The type parameter FROM_OB is restricted to be a child of OUTPUT_OBJECT (the arrow defines this so-called type *bound* and tells the compiler which operations are applicable to objects of type FROM_OB declared inside this class).

The interface lists all exported names, *repeat* is a macro installing the export list of a parent. Every parent in the parent list is typically followed by *rename, define, redefine* instructions if a parent feature is not simply inherited by the child (the default). For instance, *define* is used to announce that a feature deferred in the parent is now going to be defined. *rename* and *redefine* announce corresponding changes.

An interesting feature of Eiffel is that the visibility of child and parent is unrelated, the exports of one do not have to be a subset of those of the other. This may be surprising at first, but causes less complexity than one might expect at a first glance. A child instance still supports all the exports of its parent where it is known only to be of the parent type. This is an elegant way to combine the functionality of ADT's and inheritance to obtain what was called *compile-time checked capabilities* by Jones and Liskov in the late seventies [12].

Two parents must be disjoint or agree in the typing of their features and a feature must only be defined in either of them or stem from the same ancestor. According to our experience and taste this is one of the practical draw-backs of the visibility rules, because, as we pointed out, it usually forces the programmer to deal with every common name of two parents in the interface explicitly. In large systems, programmers experience this as

Tools	*Common Lisp*	*Eiffel*	*Sather*
Interpreter	Yes	No	No
Incremental Compiler	Yes	Yes	Yes
Debugger	Yes	No	Yes
Object browser	Yes (Inspector)	Yes	Yes
Class browser	Yes (Some)	Yes	Yes
Library source	Yes (Some)	Yes	Yes
Environment source	Yes (Some)	No	Yes
Publically available	Yes	No	Yes

by-product, most types including basic types can be specialized since the dispatching relies on type information and classes are merely one kind of type. The formal abstract data type semantics of order-sorted algebras [24], provides a pure (side-effect free) semantics for such a separation. They also base their formal treatment of subtype polymorphism on the types of all arguments. In this view, the type of a collection of arguments is the cartesian product of the single argument types, and methods may be viewed as executing 'inside' the corresponding product class. This brings back the notion of class behavior and abstract data types on a higher level.

The class 4COL-REGION below has three parents: NET, MUTEX-CONSTRAINT-MIXIN and ACTIVATION-METER and introduces five object attributes. Each attribute declaration optionally lists an initialization that evaluates and initializes the object when a new object is created. Most of the attributes also allow access to clients, here only *:accessor* functions are defined that can be used to read and write to the attribute. *:reader* functions can also be used to make the object read-only for clients, in terms of the underlying abstract data type.

```
(defclass 4COL-REGION (NET MUTEX-CONSTRAINT-MIXIN ACTIVATION-METER)
      ((red :accessor red :initform '(mk-col-unit))
       (green :accessor green :initform '(mk-col-unit))
       (blue :accessor blue :initform '(mk-col-unit))
       (white :accessor white :initform '(mk-col-unit))
       (unit-set :initform '(red green blue white))))
```

Methods refer to classes by the typing of their arguments. The method *connect-input* below will only be dispatched to if its first argument is a SILENT-INPUT-CONNECTION and its second argument is a UNIT, *and* if there is no more special method for the dynamic type of the first and second argument at the time of the call.

The second method shows the so-called *standard method combination*. With this type of combination one can add behavior to inherited methods by writing qualified methods. An *update* method will (conceptually) trigger the execution of the '*update :after*' method below when it completes its own execution.

CL compilers can implement this triggering by combining the bodies of the various qualified methods and the method proper to one *effective* method that can be further optimized. Method combination thus aims at separating type-specific part of behavior *within the clan*

references and manipulating the state as needs arise during test. A class browser displays the inheritance hierarchy and finds methods that belong to specific classes or collects the definitions that make up a single effective method. The profiler collects execution statistics for program optimization. Most implementations on workstations offer a window based interface with optional source licences for the GUI libraries. Some parallel extensions exist for multi-processors on the market or to exploit concurrency on single-processor machines, although parallel constructs are not part of the language standard. A large and vital user community exists for Lisp and is slowly absorbing CLOS. Only recently a special newsgroup continued a mailing list that existed for several years in the CLOS community. Most CLOS implementations descend from the popular public domain PCL[7] implementation.

Beside the compiler, Eiffel is supported by a class browser and a tester (inspector). The tester offers limited execution of methods since there is no interpreter. For instance, methods and their arguments can be selected if they are around, and can be run like compiled code in Lisp. Interpreters and debuggers are subject of development efforts at ISE to our knowledge. Various smaller tools exist in the environment that allow interface extractions from classes and semi-automatic generation of class documentation. Beside the basic types and those needed by the Eiffel kernel (like I/O), the class library includes several parametric data structures, classes for parsing and access to X. Also the source of the class browser is included in the Eiffel system. The Eiffel community is growing. Since about two years there is an Eiffel newsgroup.

Sather has been in operation since mid 1990 and a beta release has been publically available since mid 1991. About 500 sites have retrieved the system within the first three months and a number of ports were already completed. Because all source code is publically available, there are a growing number of general purpose library classes, program development tools and ports to different platforms running UNIX. The Sather libraries include classes implementing most of the fundamental data structures and algorithms from computer science, a variety of classes for geometry, numerical and statistics applications, as well as connectionist net simulation and user interfaces.

Sather has a rich programming environment based on GNU Emacs. The source level editing mode support syntax-oriented editing, a textual browsing facility for classes and automatic documentation generation. The Emacs debugger gdb has been extended to a Sather source level debugger, that combines instance inspection with gdb breakpoints, execution of compiled routines and other gdb facilities. Under X many functions can be invoked by a comfortable point-and-click interface to Emacs.

3 CLOS

CLOS unifies and generalizes some classical object-oriented Lisp dialects like Flavors and Loops. It is part of the dynamically typed Common Lisp (CL) language. Function calls can be dispatched. Rather than dispatching at the calling code, generic functions have an identity in Common Lisp. In fact, they are objects themselves of a generic-function class and as part of the meta-object protocol their behavior can be customized.

By making generic functions 'first-class citizens' in Common Lisp, the CL designers' goal was to merge the object-oriented paradigm with procedure-oriented style of Lisp. This provides a semantically consistent framework, in which the type character of classes, data abstraction and representation, is separated from their behavioral aspects. As an elegant

[7] Xerox Parc, Palo Alto, CA

The Boehm collector does not know about type tags and instance layout. It quickly scans certain memory areas and retains all pieces potentially referenced. Also it has the advantage that it can be used uniformly with data originating in foreign packages. However it has the 'mark-and-sweep disadvantage' of less frequent but larger pauses that make it less attractive for highly interactive programs.

The Sather libraries are designed in such a way that far less garbage is generated than is common in other systems. Instead of using linked lists as the primary container structure, the Sather libraries use the idea of amortized doubling. The primary container objects are arrays and whenever their size is insufficient the allocated space is doubled. If a container ultimately gets to be of size n, then $\log n$ such doublings will take place. The total allocated space is $1+2+4+...$ which is $O(2n)$ and there are only $\log n$ pieces for the collector to find.

Sather implements efficient arrays by allowing objects to have a variable sized array part allocated at the bottom of the object. This allows direct array access in cases where other languages require several indirections.

Q: *Is the implementation portable? What does it compile to?*

CLOS implementations rely on emitting machine code for several low-level constructs. With each port some 'machine-level patches' have to be made. However ports exist for several hardware platforms. Eiffel and Sather code compile into portable C and efficiently links with existing C code. Preliminary Sather benchmarks show a performance improvement over Eiffel of between a factor of 4 and 50 on basic dispatching and function calls. On the Stanford benchmarks[6], Sather is slightly faster than C++, though this is probably due to the C compiler's better ability to optimize for a Sparcstation than the C++ compiler.

Implementation	*Common Lisp*	*Eiffel*	*Sather*
dispatching	in generic function	call code	call code
method caching	Yes	No	Yes
inherited code duplication	No	No	Yes
type of GC	different	Dijkstra	Boehm
target code	partly machine-dependent	portable C	portable C

2.5 Tools and Environment

Q: *How is the language supported? What is covered by library classes? Is the source of library classes and major tools available? Are there public domain implementations? Is there a dynamic user community?*

Most Common Lisp vendors offer CLOS supported by the standard suite of Lisp tools, an editor, often based on Emacs, interpreter, incremental compiler, debugger, and inspector. Some implementations include a class browser and profiler (or metering tool). Usually single Lisp forms such as *defclass*, *defmethod* or *defun* can be sent from the editor to the incremental compiler and dynamically bound into the running system allowing mixtures of interpreted and compiled code as is most suitable for debugging. The debugger usually catches any errors and gives information on the program state and allows stepping through source level forms. The inspector allows the user to navigate through the data space following object

[6]including 8 queens, towers of Hanoi, bubblesort, etc.

adding a routine offset that is a function of the class index. This is implemented by another table lookup. No caching is used to our knowledge.

CLOS' dispatch mechanism supports caching. Moreover it uses various special caches that bypass standard caching such as 'one argument' dispatch (attribute access) 'two arguments' dispatch (statistically common enough to deserve special treatment), 'one index' access (the indexed attributes of perhaps different classes were accessed at the same offset). The caching scheme is described in detail in a forthcoming article[5].

Q: *Is the (binary) code for inherited methods shared or is it duplicated?*

Another set of design decisions has to do with the extent to which descendent classes duplicate versus directly use code inherited from ancestors. A similar question arises with parameterized classes. In the current Sather implementation, we decided to opt for efficiency over code size. Thus each descendent class and each set of parameterizations in a parameterized class generates its own version of the routines it uses. Because of the strong type specifications it is commonly the case that the compiler determines that many of the ancestor routines are not used and so no code is generated. In large systems we find that the most critical fundamental parameterized classes (such as LIST) appear many times (eg. a few classes appear 10 to 20 times with different parameter setting in the Sather compiler). Most classes, however, appear only once and the expansion in overall code size is not significant. Because these fundamental classes are involved so often in inner loops, the speed advantage of having separately compiled versions for different parameter values can be significant particularly due to the considerably higher potential of inlining.

Beside the sometimes increased code size, the price for this duplication is a *loss of some incremental compilation.* When a class is compiled, all its ancestors have to be read and reanalysed (*self*, at least, has a different type). Although the ancestor classes are not recompiled themselves perhaps customized versions of their methods for this class have to be compiled.

Q: *What type of garbage collection is used. What is the overhead for tagging and garbage collection?*

In languages like Smalltalk, CLOS, and Self typical programs generate a lot of garbage and the philosophy is to institute clever collectors to get rid of it efficiently. In Lisp, CONS cells are used to build up many data structures and these small bits of memory are continually being allocated and deallocated. In Smalltalk, stack frames themselves are objects that must be continually reclaimed. In such an environment, sophisticated generational scavanging collectors have made a tremendous difference. Eiffel 2.2 implements a Djkstra collector which runs almost like a coroutine with the code. Items are marked white, black or gray according to whether they have been examined, unexamined, or modified. This kind of collector has the advantage that it runs concurrently with the code and spreads the collection time over the execution. Unfortunately, it also entails marking bits on every assignment to a pointer. Also, because the code compiles into C, a separate stack of pointers into pointer variables on the system stack must be maintained. This prevents such variables from being put into registers and can be detrimental to performance.

In Sather, we chose to use a simple conservative mark and sweep collector implemented by Boehm [2], which doesn't impose any added overhead except while collections are taking place. For efficiency reasons the garbage collector is not based on the Dijkstra algorithm.

[5] Gregor Kiczales: Efficient Method Dispatch in PCL

Methods	*Common Lisp*	*Eiffel*	*Sather*
belong to classes	No	Yes	Yes
Single or multi-dispatch	Multi	Single	Single
Method combination	Yes	No	No
Method conformance	–	covariant	contravariant
Reference to shadowed methods	Yes as part of method combination	Yes	Yes
inlining	Partly	Partly	Yes
unfolding	No	No	Yes

mechanisms to speedup method lookup and object access on the average.

The Sather type system uniquely identifies basic types such as integer, real, etc. at compile time. There is therefore no need for tag bits and the associated processing overhead. Expressions operating on such types (eg. most arithmetic expressions) compile directly into analogous C code and ultimately run at the same speed as C.

All other objects have a run-time tag at the head of the space allocated for the object. This contains a unique integer that is assigned to each type by the compiler. Because the type system makes explicit the points where dispatched calls are allowed (via the "$" declarations), such tags need only be examined on dispatched calls. For these calls, Sather uses a combination of a fast hash table with local caching. Each feature name is assigned a unique integer value. The hash table hashes on the integer associated with a feature and the object tag that specifies the class. The contents of the table are interpreted as a function pointer for routine calls, as a pointer to a static storage location for access to shared class variables, as an immediate value for constants, and as an object offset for object attribute access. Only those features that are accessed in a dispatched fashion are put into the table. The table itself uses an inexpensive hashing function and collision resolution mechanism and is kept at a load factor of .5 for efficiency.

Every potentially dispatched call has two static storage locations associated with it which cache the table lookups. One of these stores the object tag of the last lookup and the other stores the value retrieved on that lookup. Dispatched calls compile into an expression that first compares the object tag with the cached object tag. If these are equal, the stored value is used otherwise a hash lookup is done and the cached values are updated. In the most expensive parts of the code (eg. repetitive loops) it is common for there to be many calls on objects of the same type. In this case the extra overhead for the dispatching is only one extra comparison.

The ability to explicitly declare object types also keeps the code size down. Because the compiler can determine more easily which routines are actually called, it can leave out more unused code. The dispatch table is far smaller for the same reason. This is particularly important in modern RISC machines, which achieve much of their performance from data caches. A smaller table improves the chances that the needed portion on any call will be in the hardware cache. As a result, many Sather run-time characteristics are close to those of hand-crafted C programs for several benchmarks [6].

Older Eiffel implementations use a (N^2) routine table in which C functions are accessed by class index plus routine offset. To avoid the required size for a large number of classes (> 1000), in more recent implementations, this table is squeezed considerably by actually

For undispatched arguments, however, Sather requires *contravariant* conformance, i.e., the parent's argument types must be subtypes of those of the child, while Eiffel requires the opposite *covariant* conformance, and CLOS does not pose any typing requirement on the undispatched arguments, such as for instance optional arguments of generic functions. Contravariance is consistent with the assumptions of parent callers. They pass data of the parent's argument type and the child's method must accept those. Only then can the compiler correctly assume that the formal (undispatched) parameters of a method have the type that is declared in the method head.

With respect to attributes, contravariance is consistent with the associated *reader* access function. It allows to specialize attribute types when redefining attributes. However, contravariance may conflict with direct assignments by clients to attributes because the underlying *writer* access function has the assigned value as an argument. Such client assignments are supported in CLOS and Sather and may assign values of the attribute type declared in a parent while the child specializes the type. The corresponding conflicts between *self* assignments to attributes in inherited code and type redefinitions of these attributes is less important. It can be resolved locally, i.e., by associating type redefinitions of attributes with redefinitions of the conflicting methods.

The choice of covariant conformance in Eiffel is better suited to using inheritance for restricting classes along the inheritance hierarchy but it invariably intertwines the semantics of dynamic binding with exception handling.[4] Every method must expect to be called with arguments of the 'wrong' type that either makes for a hole in Eiffel's static type system or requires run-time type-checking for all arguments before entering a method body optimized under the assumptions of correctly typed arguments [8]. On the other hand, it is obvious that the mathematically more pleasing contravariance, on which many formal typing systems build [3, 7], does not allow type specialization of undispatched method arguments. This often keeps argument types general for classes high up in the hierarchy. In the extreme case all undispatched arguments are of the most general type and the compiler can only use the type of dispatched arguments and there is no more distinction between a static and dynamic typing approach. We believe this clearly exposes the general trade-off between static typing and dynamic typing in object-oriented languages. *We can obtain static typing in one part of the language only by dynamically typing in another part.* This is because we cannot arbitrarily drop dynamic type-checking without risking violations of the typing assumption that the compiler makes.

2.4 Implementation

Q: *How is dynamic dispatch implemented? Is caching used to amortize the cost of dispatching?*

The choices made in the Sather implementation reflect a desire for high efficiency. Perhaps the most fundamental choice in the implementation of an object-oriented language is the mechanism for implementing dispatching. The implementation of dynamic dispatch has been subject of several papers. For instance Rose [21] compares various mechanisms including low-level interrupt based dispatching on stock hardware that makes single dispatch as fast as function calls but at the cost of portability. CLOS and Sather use caching

[4] On the Eiffel newsgroup there was also a proposal to rely on global analysis to fix the type violations possible with covariant conformance. The price however would be giving up the compositional semantics of class combination. The addition of a legal call to a parent could invalidate an existing remote descendent.

```
switch x.type
when A::type then x1: A := x; x1.foo(...); -- non-dispatched, inlining possible
when B::type then x1: B := x; x1.foo(...); -- non-dispatched, inlining possible
else x.foo(...);                           -- normal dispatch
end ;
```

classes, can treat a system and optimize it as if future extensions would not happen. For instance dispatched calls to children-less classes can be treated as undispatched. This does not hinder reuse but may require some recompilation if new classes are added later.

Eiffel, for instance, inlines non-recursive procedures that happen to be non-overloaded, i.e., procedures that have exactly one definition in the whole class hierarchy. Also dead-code elimination can optionally be called for. Eiffel can remove unused classes and unused features. This can drastically reduce the well-known burden of reusability where we drag a whole library into our programs when we use a single procedure. To our experience, this optimization typically reduces the size of executables by a factor of 2–3.

Functions are currently not inlined in Eiffel[3]. We assume this is due to the possibility of specializing functions by attributes and the desire to share code on the binary level to gain more incrementality in compilation. In Sather, functions cannot be redefined as attributes and binary code is not necessarily shared so that functions can be inlined like procedures. Additional to dead-code elimination, Sather can unfold inheritance. This includes the feature definitions of parent classes in the code of a child. While feature duplication increases the size of binary code it also goes with

- more specific type information helping the compiler in optimizing, and,
- an unfolding of inherited definitions that decreases overloading.

Particularly the last point increases the possibility of inlining, which is the basis for many other optimizations. On the one hand, this unfolding allows optimizing *self* calls and calls to classes without decendents. The parent code will not be called with *self* bound to a child instance, so that the compiler can take advantage of the child's data layout and inline many calls to *self*. On the other hand, when there are no redefinitions in existing descendents, the compiler can tradeoff sharing of a single piece of code perhaps with more dispatched calls in its body vs. duplicate versions for the different descendents.

Q: *What are the typing rules in relation to redefinition and specialization?*

The optimization and optional instrumentation of code by run-time checks often rely on typing assumptions by the compiler. Different languages take different approaches with respect to the typing rules for redefinitions. In general, we have to distinguish between the arguments that the run-time system dispatches on and the ones that it does not use for dispatch. We also must consider the method result type. No typing requirements have to be posed for arguments dispatched on, since the method looked-up fits its arguments by definition of the lookup mechanism. It is unquestionable that the result type of a redefining method must be a subtype of the inherited method's result-type since callers in inherited code assume to get back a value of that type.

[3]Function inlining is announced for a forthcoming version

Types & Inheritance	*Common Lisp*	*Eiffel*	*Sather*
Multiple inheritance	Yes	Yes	Yes
Visibility control	packages	'*export*' and '*rename*'	'*private*'
Default visibility	–	private	public
Constraints	class ordering	assertions & bounds	No
Conflict resolution	class ordering	explicit redefinition necessary	class ordering and '*alias*'
Static visibility & type checking	No	Yes	Yes
run-time type checking	optional	Yes, exceptions on violations	optional
Semantic checks	–	Class invariants pre- and postcond. included class-wise	named assertions included name-wise

Eiffel and Sather have single-argument dispatch. One argument is syntactically distinguished by so-called *dotting*: in a call *arg0.foo(arg1,...)*, the feature *foo* of the object *arg0* is applied to the other arguments. The type of this (implicitly) first argument (here *arg0*) is used to lookup a method defined in the class of that object. In this way methods belong to a specific class. The dot notation supported in these languages also suggests the metaphor of executing inside the object, and syntactically separates the one argument that is treated specially. Dispatching is implemented in these languages as part of the call mechanism.

All three languages allow extensions of parent methods without copying the inherited code to extend it and the implied danger of later parent modifications that require to update descendents consistently. CLOS supports an almost declarative mechanism for *method combination* in which methods of different supertypes can be combined to form one effective method. In Eiffel, a combination of renaming and redefinition allows one to refer to a redefined method. Given a method is both renamed and redefined then, by convention, renaming is supposed to not apply to calls of that feature. In particular the new definition itself can then call the old method under the new name. Sather's *alias* construct allows to refer to parent methods and extend them in a descendent. In Sather and Eiffel, however, method combinations is imperative by wrapping code around explicit calls to the parent methods.

Q: *What do we have to 'pay' for dynamic dispatch?*

Even if the run-time overhead for dispatching can be kept low on average, the semantics of dynamic binding make object-oriented languages much slower than their non-object-oriented relatives because function inlining is often much more difficult or not possible at all. Inlining however is the basis for many other optimization that are not possible across interprocedural calls. Making the inlined body of a function available for global optimization however can easily outweigh the cost of a more elaborate compilation scheme and of run-time type tests thrown into the code by the compiler and protecting an inlinable non-dispatched call [11] like in

Static languages like Eiffel and Sather, whose compiler deals with a closed system of

Q: *What are the rules for avoiding or resolving inheritance and/or naming conflicts?*

In CLOS programmers solve inheritance conflicts mainly by rearranging parent lists in the class hierarchy. Sometimes one has to rethink a class hierarchy and delete redundant parent orderings, or factor some functionality in different ways.

Eiffel requires one to solve inheritance conflict on a feature-by-feature basis. If parents are disjoint or share identical features originating from the same class, no conflict arises. There is also no conflict if a feature is undefined or *deferred*, i.e., without implementation, in all but one parent. All other conflicts must be explicitly solved and there is no preference rules that depend on the order of parents. This conservative approach allows more and simpler checks without forcing the compiler or human reader to look into the class bodies. In this way it promotes a separation between design (class interface) and implementation (class body) [17]. Naming conflicts that occur 'accidentally' when large projects are merged can be solved by renaming where all occurences of a name including references/calls are consistently replaced. However inheritance conflicts can only be solved by redefinition this way. One typically declares the conflicting parent features to be redefined and then copies the "right" feature over to the child. Alternatively one may rename the "right" parent feature and write a redefinition that calls it. This makes conflict resolution imperative and low-level. Imperative, since it is not possible to get by without writing a routine, despite the fact that all ingredients for the child are already there someplace in the respective parents. Low-level, because one must resolve conflicts on a feature-by-feature basis rather than "talking classes" and because the parent features that are "renamed away" are still present bearing potential for new undesired naming conflicts. So the desired high-level separation between design and implementation is weakened.

With respect to inheritance conflicts, Sather, at a first glance, takes a simplistic default approach. Sather treats parent classes as if they were included in the child class. The last definition of a feature is the active one. This is equivalent to a depth-first search in the inheritance DAG starting from the class end to the beginning. Its explanation, the inclusion/replacement metaphor, is simple and its consequences can be easily understood, but like in CLOS it also has the disadvantage that children classes have to repeat ancestor classes from time to time to 'reactivate' a bunch of otherwise shadowed features. For more complex cases, Sather offers an *alias* construct that allows to refer to an inherited feature under a different name. If conflict resolution on a single feature basis should become necessary, an inherited feature can be aliased before it is shadowed by a subsequently included class and can later be aliased back to shadow the unwanted one. Like parent class inclusion, there is a simple inclusion semantics for *alias* that is copying the referenced feature at the place of the *alias* declaration under a new name. Note that this is different from renaming, which may change the body, too.

2.3 Methods and Dispatch

Q: *What kinds of polymorphism are supported? What is the mechanism for dynamic dispatch?*

CLOS has multi-methods, i.e., dispatches on the type of multiple arguments. Overloaded functions called *generic functions* in CLOS, are objects. When called, they dispatch according to the type of all their arguments and pass them on to the type-specific method, the type-specific implementation of the overloaded function name.

While CLOS is dynamically typed, Sather and Eiffel support a combination of static and dynamic type-checking. Because of dynamic binding, object-oriented languages do not permit completely static typing. There are at least two reasons for this:

1. If attributes are specialized their types are usually specialized. Assignments to them in inherited code still reflect the more general type. Either restricting the specialization of attribute types or forcing the redefinition of all code which makes these assignments would exclude many useful forms of reuse. This problem is independent of the approach taken with respect to visibility and information hiding because it takes place only in descendent classes.

2. There are many cases in which we would like container objects like lists, stacks, or files to hold objects of several different types. Often these objects must be read out by a different class than the one that wrote them. The reader must therefore dynamically determine the types of the stored objects in order to apply only permissible functions. There are many situations such as this in which the statically declared type of a variable must be more general than the dynamic run-time type of a stored object.

Q: *Is implementation or semantics inheritance supported? Can programmers express descendent and parameter constraints?*

Descendent constraints can only be expressed in CLOS and Eiffel.

In CLOS we can constrain the set of descendents by listing the parents of a class in a particular order. CLOS demands the existence of a topological sort of the *ancestors* (the direct and indirect parents) that defines the precedence among them for inheritance [14].

Eiffel promotes *semantics inheritance*. This means children must preserve the semantics of their parents expressed in terms of class invariants, pre- and postconditions. Eiffel also supports *bounded parametric types*, this means constraints on type parameters can be expressed by listing a bounding class. Only its descendents are legal actual parameters. This allows the compiler to develop an understanding of legal operations and their typing on values of the parameter type *within* the parametric class.

In Sather, which stresses *implementation inheritance*, descendents are not constrained by the semantics of their parents. In other words, one can use inheritance to reuse and recombine the code of parent features in ways that satisfy semantic requirements different from those of the parents. Features do not carry the semantic burden of the classes they originate from. Also we have dropped bounds to shorten the learning curve and give the implementor of parametric types the freedom to refer to features undefined in the parameterized class. The Sather compiler must check the use of parameters for each actualization, a compile-time overhead that seems acceptable.

We have found semantics inheritance highly overrated in Eiffel. Since implementation cannot be separated from semantics later, when a feature is inherited, semantics inheritance excludes many useful forms of reuse. A full specification is often undesirable since the most trivial properties, like bounds for array indexes and void references for instance, quickly dominate the program text. Besides, the specification support in Eiffel is limited. The main important use seems to be run-time code instrumentation. It is doubtful whether this justifies the many related extra language features. So Sather just supports named *assert* statements. Unlike Eiffel's instrumentation which can be toggled on or off per-class, Sather programmers use the assertion names (optionally in combination with class names) like a logical proposition to classify assertions and control their inclusion.

are first-class citizens. Like in Smalltalk, the father of object-oriented languages[2], classes are objects themselves, they are *meta-objects*. These objects know to "talk" about aspects of their instances. For example, they manufacture (make-instance) instances, they allocate them, know to initialize them, to change their class, and many other details and manipulations related to the administration of objects. The CLOS MOP (Meta Object Protocol) is described in detail in [15]. Although we think that the MOP is one of the most interesting parts of CLOS, there is little basis for comparison with the other two languages in which classes are not objects themselves.

Classes	*Common Lisp*	*Eiffel*	*Sather*
include all types	No	Yes	Yes
are objects	Yes	No, though class information available	No, though class information optionally available
have shared attributes	Yes	No, but has once functions	Yes
support user-defined default initialization	Yes	No	Yes
can be created dynamically	Yes, also dynamic changes	No	No
are scoping units	No	Yes	Yes
are compilation units	1 file N definitions*	1 file – 1 class, will change	1 file – N classes
can be parametric	No**	Yes	Yes

*) *Methods and classes are separate definitions.*

**) *To avoid misinterpretations: parametric classes are often introduced in statically typed languages to model some of the polymorphic capabilities of dynamically typed languages. Although parameterized classes are not supported, parametric polymorphism is possible as pointed out above.*

2.2 Inheritance and Typing

Q: *What notion of inheritance is supported? How is it linked to the subtype relation? Is static or dynamic typing supported?*

The three languages all support multiple inheritance. They have different inheritance constraints and different approaches to typing and naming conflicts. Descendents have access to all features and there is a strong link between subtyping and inheritance. The children of a class are also subtypes of the corresponding type.

Note that for CLOS, technically speaking, classes do not inherit methods, because methods are not associated to, or encapsulated into classes. Only by explicit reference in the signature of a method to a number of argument types, methods are associated to a (cartesian) product of such types, viz., the type of a tuple of actual arguments passed to that particular method when it is selected by the dispatching mechanism. In this way methods are only loosely or indirectly related to classes in CLOS.

[2]We consider Simula the grandfather.

related to Hoare-logic specifications. In particular, classes can specify *invariants* and methods can include *preconditions* and *postconditions* to this end. The invariant must hold in all observable states, i.e., states reachable in terms of public routines only. Also routines are applicable only if their preconditions hold and they must guarantee the postconditions when they return.

In Sather all features are visible by default and *private* declarations are used to restrict visibility as needs arise and the design settles. Like in Eiffel, hiding is about clients. Descendents have unrestricted access to private features. Semantic assertions can only be associated to routines and semantics does not necessarily impose constraints on descendents (cf. Section 2.2).

Q: *Are parameterized types supported?*

Only Eiffel and Sather support *parametric classes*. This means that classes may be written with certain types left as parameters to be specified at the point of use. For example, a class implementing a stack abstraction might be defined as STACK{T}. At the point of use one might create objects with types STACK{INT}, STACK{TREE}, etc. One advantage of this approach is that the types may be known at compile time and specialized efficient code may be generated. Parameterized classes allow flexible classes to be written and used in a variety of situations without sacrificing either type safety or efficiency. In Common Lisp type declarations are optional. When types of slots are not specified in CLOS, there is no need for parameterized classes, but the attendant type checking and static optimization are lost as well. All three languages support parametric and subtype polymorphism.

Dynamic parametrization addresses the interplay between inheritance and parametric polymorphism. It allows one to specify types by reference to the type of *self* or the type of a feature. A redefinition of such a feature thus may redefine the types of several other features. In Eiffel, such types are called *association types*. Sather supports the type SELF_TYPE for this purpose. A typical usage would be for a '*create*' routine whose return type should be the same as the type of *self*. Thus SELF_TYPE is like a free type variable the substitution of which refines the meaning of an inherited method in every descendent class. Semantically, one may imagine textually copying any code or attribute definitions into a descendent class and replacing any such type references by their meaning in the descendent context. The substitution semantics of parameterized types permits implementations ranging from code sharing to specialized code per descendent class.

CLOS classes are special types. Common Lisp defines a set of basic types that are not classes and the user may define his own types via *deftype* that are distinct from classes. In particular, not all of these types can be used as specializers for methods and they do not appear as run-time tags of their objects. This situation arose because it is both possible and useful with *deftype* to define a new type that is the union of two existing types or that satisfies a unary predicate expressed in terms of the object's value or its type. In general, however, there will be no natural and unique way to fit such types into the specialization semantics required for classes. Sather and Eiffel both treat the basic types INT, REAL, etc. separately as well to avoid requiring run time tags and concomitant inefficiency. The strict typing of these languages, however, allows the compiler to determine these types at compile time. Sather and Eiffel cannot specify union types and so do not have to distinguish types from classes.

Q: *Can classes be dynamically created?*

While in Eiffel and Sather all classes are known statically, at compile-time, CLOS' classes

and Sather each object is blank at first, i.e., attributes take language-defined default values according to their types. In Eiffel, the initial state is then reached after the execution of the create routine, which must be provided by each class. In contrast, the initial state of CLOS and Sather objects is defined by initialization expressions associated to attributes (slots). An appropriate *create* respectively *initialize* behavior is automatically "composed" in the semantics of these languages, although it can be explicitly defined. This stresses the template character of classes, has a more declarative flavor and can simplify reasoning about the initial state of objects, because it seems to reduce some common types of initialization errors:

1. unbound parent attributes not initialized in descendents and leading to run time errors when methods are called on them;

2. copying parts of the parent create code with the implied danger that the children are forgotten when changes to parents are made, or that independent library users are forced to change their code appropriately before they can run it with an improved version of the libraries;

3. reference to and call of parent initialization routines and explicit reinitialization of some attributes, with the danger of becoming even more dependent on the order and interrelation of parent classes (explicit reference to their initialization routines) and the need to understand and perhaps undo their initialization (for instance in the presence of side-effects).

Q: *How are classes related to abstract types?*

Behavior abstraction in abstract data types is associated with visibility restrictions. *Clients* of a class (callers different from *self*) should rely only on the abstract interface or protocol and not on any implementation details. This enables classes to later make internal changes for portability or efficiency as long as they don't affect the interface.

CLOS has a half-hearted approach to abstract data types, because the implementation type of data often shines through in many details of the type system. In part, this is deliberate because of the wide variety of requirements imposed by different applications. For example, Common Lisp is used to implement Lisp machine operating systems and compilers for numerical applications such as S1 Lisp that are competitive with Fortran in execution speed. The required efficiency can only be reached by using implementation type information across module boundaries. Data abstraction is supported however. The Common Lisp *package* mechanism allows one to define separate name spaces. It does not however exclude accesses across package boundaries. Rather it exposes access to private names by the '::' notation. Also the data representation of classes is separated from the abstract class behavior as defined by the interface or protocol. Allocation hints to the compiler can be given in the class definition. The meta object protocol [15] also can be used to choose a specific data representation and install basic access primitives. For instance, *allocate-instance* is a method of STANDARD-CLASS and can be redefined to this end.

Eiffel stresses the semantic, abstract data type, aspect of classes and requires that semantics is inherited and must be preserved by all descendents of a class (cf. Section 2.2). Clients access objects only through their *public* or exported interface and can rely on the specified semantics independent of the particular type of object being passed. Eiffel supports many constructs for detailing the semantics of this public interface in ways strongly

CLOS, Eiffel, and Sather are just three choices in a large space of possible languages that vary along these dimensions. To assess a language, it is necessary to look beyond the language definition and to consider the whole programming environment. It is important to understand what tools exist in support of the language, whether source code is available, what libraries are available, who the other users are, etc. In our comparison we will try to consider these extended aspects of the languages besides the language definition.

We organize our comparison of the three languages by using a taxonomy defined by a series of questions. Unfortunately, often completely different terminology has developed for the very same concepts in different object oriented languages. We will introduce this terminology as we proceed. We point out the similarities and the differences between the three languages. The following table provides an initial map of technical terms for the most central concepts, polymorphic variables and functions:

Concept	*Common Lisp*	*Eiffel*	*Sather*
instance variable	slot	attribute	attribute
class variable	(class-allocated) slot	once function	shared attribute
polymorphism	generic function	no tech. term	no tech. term
class specific implementation of ~	method	routine	routine
~ with result	method	function	function
~ without result	method	procedure	procedure

2.1 Classes and Types

Q: *What are classes in the language? To what extent are they templates for instances?*

The object-oriented paradigm defines an object as a set of *features* including *attributes* (called *slots* in CLOS) and *methods* (called *routines* in Eiffel and Sather) that operate on the attributes. The three languages under consideration here are class-based, in distinction to prototype-based languages like Self (e.g. [5]). In a class-based language, a class represents the common structure and behavior of all objects that belong to it. In particular the class provides a template for dynamic instance creation. In addition, one may consider it as providing a template for the common behavior exibited by its instances. It encapsulates the procedures and functions that in some sense belong to its instances. Multi-methods in CLOS weaken this notion of encapsulation somewhat because methods need not belong to individual classes. The argument specializers of methods may instead associate a single method to a combination of several classes.

Usually an attribute may have different values in two different instances of a class. If all objects of a class *share* an attribute then it is said to belong to the class rather than the object. Here all objects in a class see the same value, and write to the same memory location.

The initial state of an object is an important semantical notion for reasoning about a program. The initial state must satisfy the class invariant, a semantic property associated to the class and valid in all observable states (between the calls to public routines). Such an invariant can be specified in Eiffel by an assertion associated to the class. In Eiffel

functions and procedures polymorphic, that is, applicable to different kinds of data. Usually two kinds of polymorphism are distinguished [4], parametric and subytpe polymorphism. Parametric polymorphisms have a single definition but their signature, that is, the type of their arguments and results, may vary. Perhaps the signature contains type variables ranging over many different types. For instance, inserting into different kinds of lists is a typical parametric polymorphism. In contrast to this, a subtype polymorphism has different definitions for different types with either the same signature or in some sense compatible signatures.

Dynamic allocation and automatic garbage collection originated in the Lisp environment and will be taken for granted by readers familiar with CLOS. Several object oriented languages, such as C^{++}, do not support garbage collection. The lack of garbage collection allows the problems of dangling pointers and memory leaks[1]. More fundamentally, forcing the programmer to explicitly deallocate memory often destroys the clean encapsulation of class abstractions. It is usually a distant caller of a class which knows when its objects can be deallocated and this often forces callers of classes to be more aware of internal structures than they should be. We therefore require garbage collection for typing, robustness and encapsulation reasons.

These four aspects of object oriented languages interact in a complex fashion. The tension between different goals leads to somewhat different notion of types.

1. There is a tension between abstract data types and inheritance. Abstract data types want to make classes be opaque boxes in which the data representation and function implementations are to be hidden. Modifying these representations then does not affect too many dependents. In contrast, a major goal of inheritance is to open the box for descendents, which want to share these implementation details with their ancestors. This saves on rewriting code and children can survive certain changes to their ancestors. Unfortunately, because of the broken encapsulation, in general, descendent classes are more easily affected by modifications to parent classes than clients are.

2. There is a tension between subtypes and inheritance. Types viewed as sets lead to a notion of subtypes as subsets of objects to which certain functions may be applied (in this way it is used for specification of legitimate operations). Among other purposes, inheritance is used both to define these subtypes and to move features of a parent class down to a child class. There are a few formal type systems that combine these views (cf. e.g. [3, 4, 24]). However they make simplifying assumptions about the uniformity of data and about the disjointness of types, which do not hold for most existing object-oriented languages. For instance, they assume that all data is tagged and that basic types are disjoint.

3. Finally there is a tension between static typing and dynamic dispatch. A primary advantages of object oriented programming is the flexibility inherent in polymorphism. This flexibility goes directly against the safety inherent in static typing. There are inherent tradeoffs in how much checking is possible at compile time versus how much must be done at run time to guarantee that, for instance, optimizer assumptions about typing are preserved during execution. These tradeoffs are reflected in both the safety and speed of exectution of programs in a language.

[1] It is well known that supposedly well tested pieces of system software contain memory leaks, causing their image size to grow without bound.

any particular language or to be especially interested in detailed aspects of language design.

The reasons for our decisions depend strongly on the requirements of these projects, but they may be applicable to other similar projects. Our experience has been that CLOS gives the more powerful and flexible environment for experimentation and prototyping but pays for this flexibility in efficiency and robustness and sometimes the loss of simplicity. It takes more development time before a program successfully makes it through the Eiffel or Sather compiler and before a prototype is up and running. It is our experience, though, that when it finally does compile, the design is closer to being 'right' and with less effort perhaps more robust and efficient than is the version developed in CLOS. On the other hand, this early guarantee of these qualities requires restrictions, which lead to loss of flexibility both in the early design and in later extensions. So, we do not prefer any one of these languages unilaterally over the others. We do not believe in single language approaches. The selection of a language depends on the goals and needs of a software project and on the skills of the people involved. Often a single language will not do the job. Special requirements and the needs of an application domain demand particular strengths and can tolerate certain weaknesses.

In this paper we compare these three languages along the major themes of object-oriented programming: classes and types, inheritance and subtyping, methods and forms of polymorphism. We also compare the available tools and environments. For each of these topics we develop an informal taxonomy and give a short tabular classification for the three languages. Finally, we illustrate and explain a few constructs from each language to provide a feeling for their varying terminologies and to motivate the reader to experiment with them himself.

Our comparison tries to be informative and concise. We assume some familiarity with the basic notions of object-oriented languages. We hope that both CLOS programmers and Eiffel programmers can get a fast overview of the language they are less familiar with. We hope the correspondences also will promote understanding of the differing terminologies used in describing the different communities.

2 Taxonomy and Overview

In this paper we will take a class oriented view of object oriented programming. In addition, we believe that for object-oriented programming to be effective, classes must provide the following features:

abstract data types, inheritance, dynamic binding and garbage collection.

Abstract data types provide encapsulation and *inheritance* eases reuse. Both promote extensibility and modifiability. These two requirements seem widely accepted and we will not address them further.

Dynamic binding of functions to function names is one key to the flexibility and extensibility of object-oriented languages. It allows new code to be called from unaltered old code by passing data of new types to old operations or assigning them to old variables. It is implemented by *dynamic dispatch* in which run-time type information is used to lookup, or bind to, the proper function. Lisp-machine operating systems typically have over 2000 classes and Smalltalk systems have about 1000. These systems have evolved over fifteen years and rely on incremental compilation and dynamic dispatch to allow extensions and reuse of the existing class hierarchies while the system is running. Dynamic dispatch makes

1 Introduction

Common Lisp [25] was developed to consolidate the best ideas from a long line of Lisp systems and has become an important standard. The object-oriented aspects developed in CLOS [1, 19, 13] were developed later and were required to be compatible with the pre-existing Common Lisp standard. This history has forced some compromise decisions that probably would not have been made had the object-oriented aspects been a part of the design of the language from the start. There probably would not be a distinction between generic functions and ordinary functions. Types probably would not be distinct from classes, and the encapsulation provided by the package mechanism probably would not have been separate from class encapsulation. Many built-in functions of Common Lisp such as hash tables, random numbers, etc. would have been separated as optional classes in a library. Partly because of these issues, current Lisp executables tend to be large and computation intensive.

The motivations underlying the approach taken by CLOS are the source of both its greatest strengths and its greatest weaknesses. Lisp has always emphasized incremental software construction and rapid prototyping. Lisp environments are among the best for this kind of work, but it has forced a strong emphasis on dynamic alterability. For example, in CLOS one can change the definition of a class in midstream. Instances that already exist must be dynamically altered before methods may be applied. To avoid the need to recompile methods when classes change, CLOS implementations typically make instance accesses independent of the instance layout. This forces extra indirections and table lookups that more restricted object-oriented languages can avoid. Such features can be quite useful, but often one has to pay a price for them even if one does not want to use them. The incremental style of design and programming that Lisp promotes is wonderful for rapid research projects or for quickly getting a first working prototype of a new and complex system. It is unfortunately often difficult to move from such prototypes to efficient, maintainable and well-debugged systems. We believe that this is due in part to the extensive flexibility provided by the environment. It is also partly due to the weak encapsulation provided by CLOS. CLOS classes do not encapsulate functionality and the strong reliance on method combination makes it difficult at times to understand exactly which pieces of code are being executed at any given time.

Our views on these issues were formed in part during the development of a vision project and of a general purpose connectionist simulation environment, called ICSIM [22]. We had used Flavors, CLOS [16] and C in other projects. A quick prototype of ICSIM was developed in CLOS. It allowed us to try out different designs but was too inefficient to develop into the final version. We rewrote and extended the system in Eiffel, a strongly typed object-oriented language [9]. Although the design was more explicit and satisfactory now, Eiffel's constraints in combining classes and reusing code led to unnecessary code repetition in several places. Also we still had problems with efficiency. In response to the needs of these and other projects, we developed Sather, a new language derived from Eiffel. Sather attempts to retain the semantically clean structure of Eiffel while achieving the efficiency of C++ [6]. We chose Eiffel mainly for encapsulation and Sather for efficiency reasons but also because we believe these languages have a shorter learning curve and the simulator is targeted for interdisciplinary projects. Researchers from widely different backgrounds must be able to develop new systems quickly using the simulator class library. We expect them to have programming experience in a mainstream language but do not expect them to be expert in

INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center Street • Suite 600 • Berkeley, California 94704 • 1-510-642-4274 • FAX 1-510-643-7684

CLOS, Eiffel, and Sather: A Comparison

Heinz W. Schmidt*, Stephen M. Omohundro†

TR-91-047

September, 1991

Abstract

The Common Lisp Object System defines a powerful and flexible type system that builds on more than fifteen years of experience with object-oriented programming. Most current implementations include a comfortable suite of Lisp support tools including an Emacs Lisp editor, an interpreter, an incremental compiler, a debugger, and an inspector that together promote rapid prototyping and design. What else might one want from a system? We argue that static typing yields earlier error detection, greater robustness, and higher efficiency and that greater simplicity and more orthogonality in the language constructs leads to a shorter learning curve and more intuitive programming. These elements can be found in Eiffel and a new object-oriented language, Sather, that we are developing at ICSI. Language simplicity and static typing are not for free, though. Programmers have to pay with loss of polymorphism and flexibility in prototyping. We give a short comparison of CLOS, Eiffel and Sather, addressing both language and environment issues.

The different approaches taken by the languages described in this paper have evolved to fulfill different needs. While we have only touched on the essential differences, we hope that this discussion will be helpful in understanding the advantages and disadvantages of each language.

*ICSI, on leave from: Inst. f. Systemtechnik, GMD, Germany

†ICSI